Copyright
by
Rohan Francis Rebello
2009

Byzantine Fault Tolerant Web Applications using the UpRight Library

by

Rohan Francis Rebello, B.E.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2009

Byzantine Fault Tolerant Web Applications using the UpRight Library

APPROVED BY

SUPERVISING COMMITTEE:

Lorenzo Alvisi, Supervisor

Michael Dahlin

To Santosh, Corinne, Sonia and Vinod

Acknowledgments

Firstly, I thank God, who works in ways beyond our understanding, but makes all things possible. I am grateful for the blessings I have been given in life — curiosity, skill and faith. Curiosity, to never stop asking questions and seek answers to them; skill, to solve the problems that I can solve on my own; faith to deal with those that I can't.

I am grateful for the blessing of a wonderful family, to which this thesis is dedicated. My parents, Santosh and Corinne have always been supportive of the choices I have made in my life. When I was unsure of whether pursuing a Master's degree halfway across the world from home, was worth the cost and effort, they reassured me that it was. They were right. Without their love and support, I would not have been where I am today. My sister, Sonia, and her husband, Vinod, deserve their share of credit – their support, advice and reassurance provided me the motivation I needed in the last few months of pushing hard to get this work done.

I thank Lorenzo Alvisi, my advisor, for the opportunity to work on this thesis and get my first-hand experience of what it means to work in the competitive environment of a leading research group. His faith in my abilities allowed me to continue working on this thesis when the odds were against me. I thank my thesis reader, Mike Dahlin, for his helpful feedback on improving the

quality of the thesis. Besides him being an amazing teacher as well, his calm and practical approach to problems has my respect and admiration. Together, Mike and Lorenzo are a wonderful team of advisors that only a lucky few will have. I hope that many more students benefit from the opportunity to work with them.

I thank everyone in LASR for the wonderful time I've spent here. Allen and Yang deserve special mention for their help and advice on the technical work of this thesis; Nalini and Prince, for their feedback on the writing of the thesis.

There are many more friends and acquaintances, whose names are too many to mention here, who have helped indirectly in their own little way, in producing this thesis. Thank you to all of you — your contribution will be remembered.

Rohan Francis Rebello

The University of Texas at Austin

August 2009

Byzantine Fault Tolerant Web Applications using the UpRight Library

Rohan Francis Rebello, M.A.
The University of Texas at Austin, 2009

Supervisor: Lorenzo Alvisi

Web applications are widely used for email, online sales, auctions, collaboration, etc. Most of today's highly-available web applications implement fault tolerant protocols in order to tolerate crash faults. However, recent system-wide failures have been caused by arbitrary or *Byzantine* faults which these applications are not capable of handling. Despite the abundance of research on adding *Byzantine fault tolerance* (BFT) to a system, BFT systems have found little use outside the research community. Reasons typically cited for this are the difficulty in implementing such systems and the performance overhead associated with them. While most research focuses on improving the performance or lowering the replication cost of BFT protocols, little has been done on making them easy to implement.

The goal of this thesis is to evaluate the viability of BFT web applications and show that, given the right abstraction, it is viable to build a Byzantine fault tolerant web application without extensive reimplementation

of the web application. In order to achieve this goal, it demonstrates a BFT implementation of the *Apache Tomcat* servlet container and the *VQWiki* web application by using the *UpRight* BFT library. The UpRight library provides abstractions that make it easy to develop BFT applications and we leverage this abstraction to reduce the implementation cost of our system. Our results are encouraging — less than 2% of the original system needs to be modified while still retaining all the functionality of the original system. Given the design trade-offs that we make in implementing the system, we also get comparable performance, indicating that implementing BFT is a viable option to explore for highly-available web applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of this thesis is to evaluate the viability of Byzantine fault tolerant (BFT) web applications, given the abstracted implementation of a BFT protocol. We achieve this goal by implementing a Byzantine fault tolerant wiki application using the UpRight library [7] on the Apache Tomcat [34] servlet container and the VQWiki [38] web application.

In today's large scale distributed software systems, failures of individual components of such systems can potentially affect the rest of the system to the point of making the system unavailable. Depending on the severity of the failure, this unavailability may be temporary [2, 33] or permanent, even causing loss of data [4].

Since any unavailability is undesirable, contemporary systems that are designed for high-availability, implement fault tolerant communication protocols between system components in order to handle component failures. Most such systems [6, 11, 44] implement protocols that tolerate a bounded number of fail-stop faults (it is assumed that the failing component simply crashes and does not interfere with the functioning of the rest of the system).

However, this type of fault tolerance has shown to be insufficient in

practice due to the presence of arbitrary failures. Such failures, otherwise known as *Byzantine* failures [23], can potentially cause the entire system to be unavailable [2, 4, 33].

Over the last decade, several Byzantine fault tolerant protocols have been developed to tolerate Byzantine faults [1, 5, 8, 10, 21, 22, 36, 41, 42]. Despite the abundance of implementations that provide good performance, low replication cost and robustness against malicious behaviour of failed system components, BFT protocols have not been widely adopted outside of the research community.

The UpRight project [7] aims to dispel the concern that BFT protocols are not useful in practice because of the high performance penalty and difficulty of implementation associated with them. It demonstrates Byzantine fault tolerant implementations of the Zookeeper [44] and HDFS [16] cluster services and argues that the UpRight BFT library makes it easy to add Byzantine fault tolerance (BFT) [1] to cluster-based services by minimizing changes to the original application. Their results show that despite concentrating on the ease of adopting BFT, performance does not suffer much [7].

While the UpRight project concentrates on applying BFT to cluster-based infrastructure services such as Zookeeper and HDFS [7], in this thesis we concentrate on applying BFT specifically to web applications by using the UpRight library.

[1]The letter 'T' in the acronym BFT is used to denote both the adjective *tolerant* and the noun *tolerance* depending on its context of usage

Very little work has been done to provide Byzantine fault tolerance to web applications (refer to Chapter 2). This is because implementing a BFT web application is not an easy task and requires some design trade-offs. We identify the following obstacles to the widespread implementation of BFT from the point of view of web applications:

- Web application servers are multi-threaded in order to execute requests in parallel — the order in which requests are executed depends on the order that the threads handling them are scheduled. Since thread scheduling is non-deterministic, the order of request execution could vary across server replicas. The most widely used fault tolerance approach — state machine replication [31] — requires that requests be executed in the same sequence across all server replicas. In order to support state machine replication, web application servers need modifications which either result in low performance or complex implementation (refer to Chapter 2).

- Web applications are typically designed as multi-tiered systems [28]. Since each tier implements its own fault tolerance protocol, it is difficult to implement an end-to-end fault tolerance protocol across all tiers — customized implementations of BFT protocols may be required, one for each tier [25, 36]. This increases complexity and consequently, the amount of effort required to make the system Byzantine fault tolerant.

- Web applications are used with a wide variety of web clients and browsers. Thus, a viable BFT solution for web applications must support all pos-

sible web clients in order to give the end users flexibility to use the client of their choice. Given the variety of web clients and browsers, modifying each one of them in order to support BFT is clearly not a feasible task. Therefore, any BFT solution must involve minimal changes, if any, to the web client. We consider this fact when we design our client adapter, so that it can be used with any web client without any modifications to the client (Refer to Chapter 6).

In this thesis we present a Byzantine fault tolerant web application —the VQWiki [38] wiki application, running on the Apache Tomcat servlet container [34]. Our results show that the addition of Byzantine fault tolerance requires reimplementation of a very small (less than 2%) portion of the existing application while abstracting out implementation details of the BFT protocol to the UpRight library, making this a feasible option to consider when implementing high-availability web applications. Performance is also comparable for certain workloads, without affecting usability. Also, the reusability of our BFT Tomcat server makes it easier to build more web applications on top of the Tomcat server.

1.1 Contributions

This thesis we make the following contributions:

- We provide a Byzantine fault tolerant web application - the VQWiki web application [38], deployed on the Apache Tomcat servlet container [34].

We utilise the UpRight library [7] in order to add Byzantine fault tolerance, and find that it simplifies our implementation by abstracting the details of the BFT protocol.

- We provide a reusable framework to build other BFT web applications on the Apache Tomcat servlet container using the modified version of Tomcat and the UpRight library.

- We provide lessons learnt from implementing state machine replication on the Tomcat server, based on the challenges we faced.

The thesis proceeds as follows: Chapter 2 highlights the state of the art with respect to Byzantine fault tolerant web applications. Chapter 3 gives an overview of the UpRight BFT Library, and the interface it provides. Chapter 4 details the technical background in developing the BFT Web Application system. This includes descriptions of the Apache Tomcat servlet container, the VQWiki web application and the web browser that forms the client of the system. Chapter 5 outlines the requirements of the UpRight library that pose challenges to implementing BFT versions of Tomcat and VQWiki. Chapter 6 and Chapter 7 detail the modifications made on the client side and server side, respectively in order to support the UpRight library. An evaluation of our approach to implementing BFT is presented in Chapter 8, along with steps that can be taken to improve the performance of the system. Chapter 9 presents the proposed architecture for multi-tiered systems and Chapter 10 summarises the conclusions drawn and lessons learned in this thesis.

Chapter 2

The State of The Art

This chapter deals with the related work in the area of Byzantine fault tolerance related to web applications. We find that existing systems provide some but not all of the building blocks necessary to build an end-to-end Byzantine fault tolerant web application with minimal effort on the application designer's part. First we look at the existing fault tolerance provided by Tomcat. We then see that the state machine replication approach used by the UpRight library places requirements on Tomcat that require modification of the codebase. Finally, we note that although BFT has been implemented for individual tiers in multi-tier services, no end-to-end BFT solution has been implemented for such services.

2.1 Existing fault tolerance in Tomcat

The Apache Tomcat server provides load balancing and crash fault tolerance through fail-over clusters [34]. If a particular server in the cluster fails, all subsequent client requests are processed by the other servers in the cluster. This mechanism, however, does not tolerate Byzantine failures among the servers in the cluster. It also does not provide fault tolerance to the

application hosted on top of Tomcat - the application is expected to implement its own means of fault tolerance and checkpointing.

2.2 Implementing BFT - The state machine replication approach

State machine replication [31] is a popular approach used to implement fault tolerance [5, 8, 21, 22, 36, 41, 42] and is the method used by the UpRight library [7] to implement BFT. However, using the state machine replication approach requires that requests be executed in the same sequence across all server replicas, which raises issues with multi-threading [7].

Most web application servers implement multi-threaded request handling to maximise performance. However, this provides no guarantees on the order that concurrent requests will be executed on the server due, to the non-deterministic nature of thread scheduling. In order to implement BFT using state machine replication, this non-determinism due to multi-threading, must be removed. There are three ways this can be done — (1) all requests must be processed serially by a single thread; (2) thread scheduling must be made deterministic and identical across all replicas; (3) requests are allowed to execute in parallel, but are synchronized only when they modify the same part of application state. We elaborate each of these approaches in the following sub-sections.

2.2.1 Single-threaded execution

A naive solution to this problem is to execute requests in serial order. For a multi-threaded application, this means reducing the performance advantages gained through parallel request execution. However, we observe that it is possible to take advantage of the optimization of *read-only requests* [5, 7] to implement multi-threaded read-only request processing, while maintaining a single thread to execute requests that have been ordered by the agreement protocol.

We implement this *single-threaded write/multi-threaded read* execution in Tomcat as it is the simplest approach. As the results in Chapter 8 show, this decision does not adversely affect performance in our chosen wiki application for certain workloads, because of the internal serialization of requests in the web application itself.

2.2.2 Deterministic thread scheduling:

Some work has been done to provide deterministic thread scheduling in the target application platform [3] in order to implement a BFT Apache HTTP web server. However this work relies on the assumption that messages sent between replicas are guaranteed to be delivered reliably in the order they were sent. In order to provide such an abstraction of a reliable, ordered communication system, more implementation is required on the application developer's side, increasing the complexity of implementing BFT. By contrast, the UpRight library aims to make the implementation of BFT simpler by abstracting

all the replication and protocol details away from the target application [7].

For Java applications, the Java Virtual Machine (JVM) provides a possible location to implement deterministic thread scheduling. Fault tolerant versions of the Java Virtual Machine have been built [13, 29] but they do not handle Byzantine faults.

2.2.3 Allowing parallelism for independent requests

Another option for ensuring deterministic multi-threaded execution across replicas is to allow parallelism in request execution, but restrict it to requests that do not conflict in the portion of state that they modify. Kotla et al. [22] have implemented this on by partitioning requests into non-conflicting sets based on rules specified by the application designer. These non-conflicting sets of requests can be executed in parallel while producing the same result. However this requires a priori knowledge of request conflicts which is not always possible — in a web application that uses a database for storage of state, it is not possible to have this information before executing the requests to identify conflicts [36]. A solution to this is to have conflicts identified by executing the requests on one replica and having this information passed on to the other replicas so that they can schedule requests in the same order [36].

While this strategy is certainly promising, implementing such a scheme is not easy and requires a large engineering effort. Also, despite the multi-threaded nature of the web application server, the web application itself may perform coarse-grained serialization of the requests issued to it, thereby de-

feating the performance advantage gained from parallelism and rendering such a complex implementation unnecessary.

2.3 Fault tolerance in multi-tier systems

The Thema library [25] facilitates the development of multi-tier BFT web services. While it provides a framework for implementing BFT at an individual tier, we are more interested in providing end-to-end BFT. Thema also assumes that all tiers of the application use the same communication protocols and can thus reuse the Thema library. Web applications, however, do not always use web service protocols between application tiers.

For example, consider a 3 tier web application consisting of a web client, application server and database. The web client and application server communicate through HTTP [12] while the application server and database communicate through a protocol that is custom to the database. Further, while Thema can be used to implement BFT in every application tier in order to provide end-to-end BFT, this could possibly result in more complex implementation. This is discussed further in Chapter 9.

2.4 Existing BFT web applications

To the best of our knowledge there are no implementations of BFT web applications. BFT HTTP web servers have been implemented [3, 41], but these are either experimental or do not support web applications.

2.5 Summary

Thus we find that existing systems do not provide all the building blocks needed to implement BFT in a web application. Present web applications implement only crash fault tolerance through an approach that is incompatible with the approach taken by BFT protocol implementations. There are means of addressing this incompatibility, but they result in extra complexity. In this thesis we address this incompatibility by using the UpRight library and making simple modifications to the server, gaining the advantage of simplicity of implementation by leveraging the abstractions provided by the library.

Chapter 3

The UpRight BFT Library

In this chapter we present a brief description of the UpRight library [7] the system architecture envisioned by it and the interface it exposes to the target application. We also provide a brief overview of the theoretical model assumed by the UpRight library and describe its implications for the web application using it.

3.1 Overview

The UpRight library [7] is a high-availability library used to provide fault tolerance to applications through state machine replication [31]. To this end, the library manages the client-server communication and server replication protocols entirely. Figure 3.1 describes the high-level architecture of the UpRight library. The UpRight system consists of a replication core and client and server libraries to connect the target application to the core. The replication core comprises a *Request Quorum* stage and an *Order* stage, the details of which can be found in [7]. The library can be configured according to the level of fault tolerance (crash or Byzantine) expected from the system [7]. For the purpose of this thesis, the library is configured for Byzantine fault tolerance.

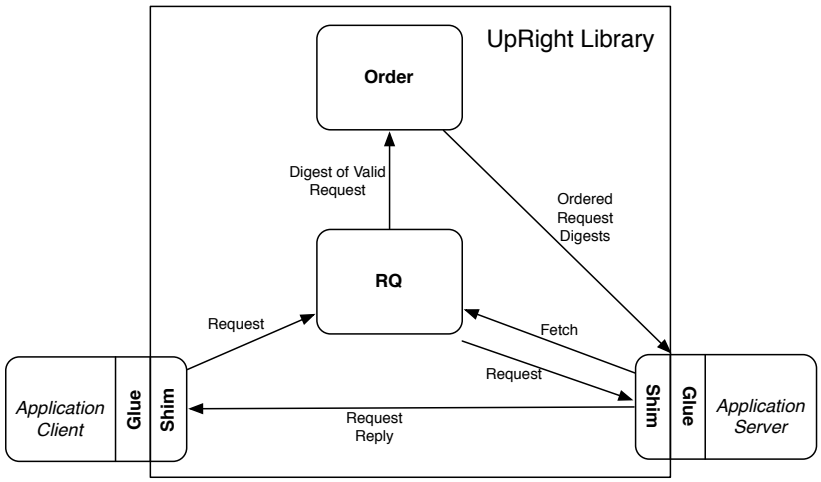

Figure 3.1: The UpRight system architecture (source: [7]). Direct client-server communication is completely replaced by a custom protocol through the client and server libraries.

The library is intended to be minimally invasive with respect to modifications made to the application it is to support [7]. To this end it provides an interface both on the client and the server side, that is to be implemented in an application-specific way, to transfer requests and responses, and to take periodic application checkpoints [7]. To better understand the system, a couple of terms are used widely in this thesis:

Execution — The *Execution* stage as defined in [7] is used to refer to the server-side part of the service and consists of the Execution shim, the Execution glue and the Application.

Shim — The *Shim* as defined in [7] is the component of the UpRight client or server library that implements generic functionality common to all

target services, namely server replication and communication with the other components of the library. The client shim handles the collection of the required quorum of server responses and delivers the response when the required number of matching responses (according to the replication protocol) is received. The server shim in the Execution stage receives requests from the Order stage and hands them to the server glue to be executed. It passes on checkpoint tokens to the Order nodes to enable them to garbage collect their logs and checkpoints [7]. It is also responsible for the transfer of application state from one server replica to another in the event that the receiving replica needs to recover from a failure. It exposes an external interface to the client and server side, to be implemented by application-specific code, known as the glue.

Glue — The *glue* as defined in [7] is the application-specific part of the client and server sides of the system, that acts as an adapter between the interface exposed by the shim, and the original client and server code of the service. The glue acquires significance only in the adaptation of an existing system to the library, since a system written from scratch can simply implement the shim interface itself. The server glue is responsible for maintaining checkpoints and loading them if instructed to do so through the shim interface [7].

Application Snapshot — An application checkpoint is otherwise referred to as an *Application Snapshot*. It typically is coarse-grained and involves

taking a snapshot of the entire working state of the application. For the Tomcat and VQWiki web application, this working state is the combination of the client session information stored in Tomcat, and the working state of the VQWiki application, as stored in the data store.

Checkpoint — A *checkpoint* refers to both the coarse-grained *application snapshot* as well as the finer-grained incremental checkpoint [7]. In this implementation, a *checkpoint* consists of an *application snapshot* and zero or more delta checkpoints each consisting of logs of requests executed for the corresponding interval between successive checkpoints.

Checkpoint Token — A *checkpoint token* is a verifiable digest of the entire checkpoint. Its implementation is application-specific and its purpose is to represent the entire application checkpoint within the UpRight library to enable the Order nodes to garbage collect their logs and checkpoints [7]. Since it is expected to be small compared to the entire checkpoint, this makes for faster processing within the Order nodes. It is also used by the shim to instruct the glue to load a particular checkpoint in the event of recovery of a failed Execution node or catching up of a slow one. As described in section 3.2.2 it is also used by the glue to verify the pieces of state given to it by the shim during recovery.

3.2 Interface Description

This section gives a brief overview of the interface provided to the application by the UpRight client and server libraries.

3.2.1 Client interface

The interface exposed to the client by the client shim is simply to pass on a request to be executed through the library. For compatibility, the request is represented as a byte array [7]. This raises scalability issues with stream-oriented requests and responses in systems with large-sized requests and responses. For these types of systems it may be necessary to have an alternate scheme to transfer the large-sized requests and responses. We do not explore these issues, however, and treat them as beyond the scope of this work. We therefore do not assume requests and responses beyond a few megabytes in size.

The request execution interface currently exposed to the application client code by the UpRight library is a blocking call in order to safeguard the system from performance degradation due to faulty clients [7, 8]. This presents a performance problem for web browsers which typically make multiple requests in parallel. A possible performance optimization here would be to have a non-blocking call to the UpRight library to allow multiple requests to be issued in parallel. Alternatively, the client glue can be implemented to include multiple UpRight clients, in order to allow one client application to execute multiple requests in parallel. This is beyond the scope of our imple-

mentation but is an interesting option for future work aiming to increase the usability of the UpRight library for web browsers.

The client shim collects responses from each of the server shims through the UpRight library and delivers the server response when a quorum of matching server replies are received. In the case of the read-only optimization [5] all the server responses are returned in order to allow the client glue code to canonicalize the response [7].

3.2.2 Server interface

The server shim interface exposed to the client is divided in to two parts each handling a single direction of control and data flow between the shim and glue. This is because each of the calls made to the glue by the shim are expected to be non-blocking and thread-safe, and thus a callback interface is provided for the glue to return server responses and application state to the shim.

3.2.2.1 Interface to be supported by glue

The following functions are to be implemented by the glue.

- Execute the given batch of requests in the order supplied in the batch (refer to Section 7.2.1).
- Take a checkpoint of the application's working state and return a verifiable token of this checkpoint (refer to Section 7.2.1.1).

– Load a particular checkpoint given the token and the data of the checkpoint (refer to Section 7.2.2.1).

– Fetch a piece of checkpointed state that might be requested from another Execution node. This is used to assist another server to recover from the state fetched by this one (refer to Section 7.2.2.1).

3.2.2.2 Interface exposed to glue

The shim exposes the following API for the glue to respond to requests issued to it:

– Return the response from executing a client request. Depending on whether the request was a read-only request or not, return the corresponding response type (refer to Section 7.2.1).

– Return the checkpoint token as a result of taking a checkpoint (refer to Section 7.2.1.1).

– Return the checkpoint state that was requested earlier by the shim (refer to Section 7.2.2.2).

– Indicate to the shim whether the glue is ready to receive requests or not. This enables the shim to throttle the requests that it sends to the glue. We do not use this functionality in our implementation. Instead requests are queued within the glue when they cannot be immediately processed (refer to Section 7.2.2.1).

While this interface is straightforward, it hides a significant amount of design issues and requirements. As seen in Chapter 5, Tomcat and VQWiki do not necessarily meet all of them.

3.3 Theoretical Model

The theoretical model of the UpRight library is described in [7]. Since the VQWiki and Tomcat web application uses the UpRight library for server replication and communication between clients and servers, it assumes the same model as the UpRight library [7]. Consequently, the guarantees for safety and liveness in the system directly follow from the safety and liveness guarantees of the UpRight library.

In existing web applications clients and servers communicate using HTTP [12] which operates over TCP [30]. The client-server protocol therefore assumes reliable and ordered message delivery — messages are always delivered and in the order that they were sent. The UpRight library on the other hand assumes an unreliable network that may omit, modify, delay or reorder messages sent through it [7]. However, the library hides this unreliable behaviour and exposes a reliable and ordered message delivery abstraction to the client and server, which fits well with the requirement of a web application, while at the same time adding tolerance for Byzantine faults.

Chapter 4

The Original Web Application

This chapter provides an overview of the existing web application to which we add Byzantine fault tolerance. It begins with descriptions of Apache Tomcat and VQWiki applications and concludes with a description of the client of the system.

4.1 The Apache Tomcat Servlet Container

Apache Tomcat [34] is an open source software implementation of the Java Servlet [32] and JavaServer Pages (JSP) [20] technologies[1]. It is designed to receive incoming HTTP requests, process them using servlets that are executed according to a set of configured rules, and return HTTP responses to clients. It consists of various components that play different roles in the processing of requests.

Catalina — This is the servlet container that implements the Java Servlet [32] and JavaServer Pages [20] specifications. The data in the request is used

[1] Java, and JavaServer Pages are trademarks or registered trademarks of Sun Microsystems, Inc.

to choose the servlet required to process the request and return the response to the client.

Coyote — This is the HTTP connector that implements the HTTP 1.1 protocol [12]. It listens at the configured server port for incoming TCP connections from HTTP clients and hands the connected socket to the Catalina container for request processing.

Jasper — This is the JSP compilation engine that compiles JSPs from xml documents into Java classes to be used as servlets by the Catalina engine to process requests.

For this work, Tomcat was chosen among many alternatives for Servlet container implementations [14, 15, 19, 40]. All these servers are implemented in Java, which fits well with the UpRight BFT Library [7] which is also written in Java. Tomcat's popularity, modularity and stable codebase made it a convincing choice as a demonstration of making BFT Web Applications a reality.

A point to note about servlet containers in general is that in fulfilling the Servlet specification [9], client session information is stored and managed within the servlet container. Web applications can also access and modify this session information based on any client request using the session API [34], and thus it is theoretically possible for any request issued to a web application to result in the modification of this information on the server. This possibility has significant consequences in the types of assumptions to be made about

read and write requests when designing a generic BFT servlet container, since every request is potentially a write request. Knowledge of the web application is required in order to be able to verify that a request is read-only with respect to server working state, implying a customization of the client glue for the target web application (see Section 6.2.1 for more detail).

4.2 The VQWiki Web Application

VQWiki or VeryQuickWiki is "Wiki server software written using JSPs and Java Servlet technology" [38]. It enables the creation and editing of many interlinked pages, referred to as *Topics*, and allows users to add attached content such as images and other files in addition to text on each topic. It is a servlet application intended to be deployed on a servlet container such as Tomcat and can be configured to use either the file system or a database as its data store to store all user content. It maintains all its application state in the chosen data store. However, client session information is managed exclusively by the servlet container, which in this case, is Tomcat. Consequently, the entire state of the web application can be viewed as a combination of client session information in Tomcat and the state managed and stored by the VQWiki application.

Among the multiple choices of open source wiki software, VQWiki was chosen because of its stable codebase, active development and use of file system as data store. For the purpose of proving the concept of a BFT web service, this proved simpler than using a database as a data store, since there is only

one tier in the application. It is worth noting that the principles applied for a single-tier service can be extended to multi-tiered services as well — this is an opportunity for future work (refer to Chapter 9).

4.3 The Web Browser Client

The client of a web application is typically a web browser. The browser makes HTTP requests through a TCP connection to the web server (in this case Tomcat), and receives HTTP responses through the same TCP connection. In the HTTP 1.1 protocol [12], multiple HTTP requests and responses may be sent along the same TCP connection. According to the protocol, the browser may choose to send an HTTP request along an existing TCP connection to the server (if one exists) or create a new TCP connection to the server. In either case, the server sends the response along the same TCP connection that the client made to it. For the purpose of this thesis, we do not assume the use of any specific browser as a client, since our approach is applicable to all HTTP clients (refer to Chapter 6).

For the Tomcat and VQWiki web application, the client is a web browser that issues HTTP requests and receives HTTP responses. A design choice to be made here is in the location of the client glue code. Ideally the glue code would be located in the client code itself, since it is tightly coupled with the client. However in the case of a web browser this is not a straightforward approach, given the variety of browsers and platforms that exist as web clients. Any BFT client would have to work on all browsers that the original

web application supports. JavaScript [2] is a scripting language that presents itself as a possible option for implementing the BFT functionality directly in the browser. However JavaScript has compatibility issues across web clients — most browsers execute JavaScript based on different standards, and some basic clients such as PDAs or mobile phones do not execute JavaScript at all. Further, complex implementations of JavaScript require extensive debugging and testing to verify their compatibility on all browser platforms.

An alternative would be to make the client glue a separate application that the browser can communicate with as if it were a proxy HTTP server. This solution is preferable in a heterogeneous browser environment where it is infeasible to create a client glue for each browser. As explained in Chapter 6, we choose to implement this solution in keeping with the principle of making the implementation as simple as possible. We also note that the glue can be implemented as a Java Web Start application, making this an attractive area for future exploration.

[2] JavaScript is a trademark of Sun Microsystems

Chapter 5

Requirements of the UpRight library

In this chapter we list the requirements of the UpRight library that need to be met an application it is replicating for fault tolerance. We find that these requirements pose challenges in using the library to implement a Byzantine fault tolerant Apache Tomcat and VQWiki.

5.1 Checkpoint generation

The UpRight library assumes that the target application has some mechanism to provide checkpoints of its state to the library. While this is true of applications like Zookeeper and HDFS [7], in Tomcat this is not the case. Tomcat possesses no such mechanisms to generate a checkpoint of its state. It does however possess a mechanism to save client session information upon graceful shutdown of the server, and to restore this information upon server restart. Unfortunately this mechanism is of limited utility as-is, since it is desirable that the checkpoint be generated while the server is running, and preferably without halting request processing for the purpose ensuring quiescence of the system [7]. Either way, the client session information is usually far from being the complete state of the web application and here Tomcat

exposes no mechanism whatsoever to checkpoint this state. The application hosted on Tomcat is expected to autonomously handle checkpoints, without any coordination from Tomcat. This makes it harder to implement deterministic checkpoints that include the entire state of the server, since this requires an implementation from checkpointing within Tomcat, and explicit coordination between Tomcat and the checkpointing mechanism, if any, in the web application in order to capture the entire working state in the checkpoint. Since neither of these is available, we implement application checkpointing from scratch. This may seem to drive up the cost of implementation, but as seen in Chapter 7, the checkpointing mechanism we develop for Tomcat, can be used as a building block to build other BFT web applications on top of Tomcat.

5.2 Deterministic Checkpoints

The UpRight library requires that each of the replicas in the replicated application server produce identical checkpoints [7]. In order to provide identical checkpoints to the UpRight library, each of the server replicas must execute identical sequences of requests [7, 31], or, in other words, in the same sequence that is provided to them by the UpRight library.

For Tomcat this is not completely straightforward as Tomcat uses a multi-threaded request processing model to execute requests in parallel, without providing any guarantees over the sequence of request execution. Hence, one of the challenges to be overcome in this work is to make the execution

of requests deterministic and identical across all server replicas. We overcome this by serializing request execution through single-threaded request processing thread (refer to Section 7.3).

5.3 Deterministic Request Execution

A standard requirement of the UpRight library is that given the same starting state, the replicated servers generate identical responses for the same request [5]. For the HTTP protocol, this presents some interesting issues:

Timestamp of response — The HTTP protocol [12] specifies a `Date` header field that can be a part of a server response. Under normal circumstances this would be based on the real time at which the response was generated on the server. However, because in an asynchronous system no assumptions can be made about the relative speed of clocks or the time at which a request is processed at each server replica, the virtual time as provided by the UpRight library [7], is used for this purpose.

Cache-control timestamps — Another artifact of the HTTP protocol [12] is cache-control, intended for caching of frequently fetched data at various points in the network path between the client and server. In order to support this, when serving content such as files, the server includes a timestamp in the HTTP response indicating when the file was last modified. Typically for images and other binary files that remain largely static on the server, this is implemented in Tomcat as a direct mapping

of the file's last modified date as recorded by the file-system. In order for the responses to be identical, these timestamps must be identical. We solve this issue by ensuring that the file's last modified date attribute on the file-system, is the same across all replicas.

JSP Session ID — An artifact of Java Servlet technology is the use of a session id [9] to uniquely identify a client and access client-specific state stored on the server. The creation of this id on the server involves the use of as much randomness as is available in the individual Java Virtual Machine (JVM) and file system on which the server is running, in order to produce a securely random id. Unfortunately this means that the generation of the session id is non-deterministic. In order to use the UpRight library correctly, each server replica needs to deterministically create the same session id given the same request, which potentially reduces the security of the session ID. However, the UpRight library provides a mechanism to use a seed value that is supplied with each request to be executed, and is the same across all server replicas. The resulting randomness of the session ID will be only as random as this supplied seed value.

Chapter 6

Client Side Adaptation

This chapter details the modifications made to the client side of the client-server request/reply flow in order to adapt a web application to BFT using the UpRight library. It highlights an important design consideration when implementing the client side of the UpRight library and describes the approach taken in this thesis.

6.1 Design choice - Client module vs Proxy server

As mentioned in Chapter 4 the intended client for a web application is a web browser and an early design issue is whether to include the BFT client shim code as part of a browser, or to have it run as a proxy server to the browser. While there is no clear winning solution, each has its merits and demerits:

6.1.1 Client Glue as Custom protocol plug-in

In this approach, the client glue is implemented as a module of the browser itself, possibly as a plug-in. The Mozilla Firefox browser [27] for example, allows this type of approach. This has the advantages of being more

tightly integrated with the browser client and also more usable, since this avoids the need to run a separate process for the client shim. Such an approach would be best suited to a relatively naive end-user audience since it offers easy adoption — "simply install an updated version of your browser", would be the instructions to a user of the BFT web application.

However, this approach is not as simple as it seems — it may require a customized implementation of the UpRight client library itself. Currently the UpRight library is available as a Java implementation [7] which might not be compatible with all browser implementations. In a large heterogeneous browser environment where it is required to support a wide variey of different browsers for a system, the costs to develop a custom implementation for each browser outweigh the benefits of having tighter integration. Further, having multiple browsers and platforms share a single BFT client is becomes attractive if the expected per-client request throughput and latency requirements are low.

An alternate approach to this solution can use JavaScript to implement the client glue. This also has the advantage that the end user will not have to install and run a separate proxy process as is required in the traditional proxy server approach (see 6.1.2). However, JavaScript is known to have cross-browser compatibility issues — different browsers execute JavaScript according to different specifications, and some clients such as mobile phones and PDAs may not support JavaScript. This lack of cross-client compatibility poses a significant usability challenge to a JavaScript implementation of the client glue.

6.1.2 Client Glue as Proxy Server

As shown in Figure 6.1, with the client glue implemented as a proxy server, the browser makes HTTP connections to the client glue instead of to the server. This approach has the advantages of being compatible with all browsers and results in a simpler one-size-fits-all implementation of the client glue. However, the question arises as to how many browsers the client glue should simultaneously support. Since the UpRight client serializes the requests sent through it and executes them one at a time, supporting too many browsers on a single client glue would result in performance and latency issues for the browsers. Ultimately, the choice of how many clients to support using a single client glue remains dependent on the latency and throughput requirements of the clients of the web application. For our implementation, we use a single client per glue to maximise per-client performance.

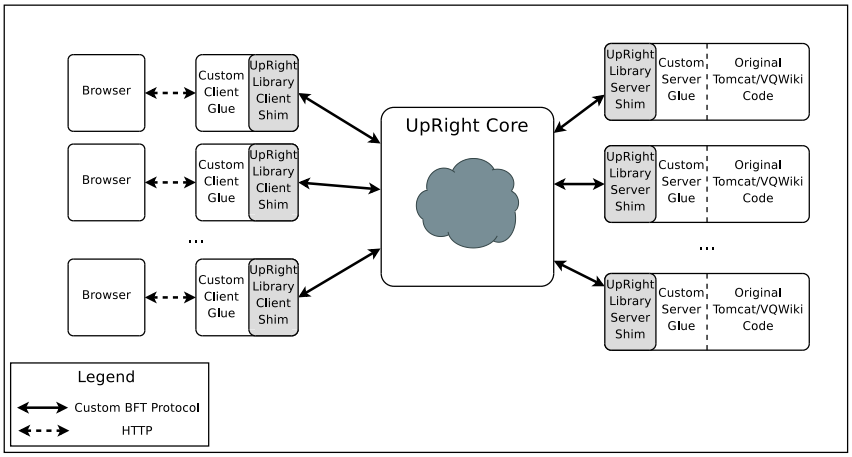

Figure 6.1: The proxy server approach to implementing the Client glue. The glue behaves like a proxy HTTP server to the browsers that connect to it. For simplicity we show one client glue per browser.

In this thesis we choose the proxy server implementation in order to keep the implementation cost of the system low, since only one implementation of the client glue is required to support all web clients. The proxy server is implemented as a Java application utilising the UpRight client library to provide the abstraction of an HTTP server to the browser that connects to it as shown in Figure 6.1.

6.1.2.1 Java Web Start Client

An alternate implementation of the client glue as a proxy server, is as a Java Web Start application [18]. Whenever the user wants to use the system, they download the web start application through the browser. Once the download is complete, the application is automatically run on the Java Runtime Environment (JRE) of the user's system. This solution removes all the overhead of having to install and maintain a standalone proxy server application and allows the client glue application to be distributed to users in a convenient manner. While implementing this solution is beyond the scope of our present work, we believe it is a fruitful area for future work from the point of view of usability.

6.2 Utilizing the UpRight Client Interface

The UpRight client library exposes an interface to execute a request specified as a byte array for compatibility with any application. Further, the mechanisms of the library prevent the client from making more than one re-

quest simultaneously. This effectively serializes all requests that are made through a particular UpRight client which has important performance consequences on the applications that utilize the client. Typically a browser opens multiple TCP connections to the server in order to execute the usually multiple HTTP requests that arise from loading images, scripts and other content on a typical HTML page. The serialization of requests could potentially lead to a degraded browsing experience with pages involving many images, since these will all be loaded serially and not in parallel as would occur in a normal browser. However, we find that since such multiple requests are issued only after the main HTML page has loaded, this does not have a significant impact on the browsing experience when using the UpRight library.

6.2.1 Read-only requests

The UpRight library supports the use of read-only requests that bypass the request ordering protocol [7]. This avoids the overhead of the agreement protocol being added to the request execution time and has shown to improve throughput in systems that employ it [7].

In a web application, this optimization is straightforward for requests that access static content on the server such as images and files. However, for dynamic content that is generated by a servlet for example, implementing this optimization requires internal knowledge of the working of the web application to determine if the code in the servlet would modify the web application's working state, or simply generate output wihtout any side-effects.

Despite the internal knowledge of the servlet application, side-effects are still possible with requests that are perceived as read-only from the point of view of the servlet application. For example, the use of client sessions as defined in the Servlet specification [9] results in every request issued to a servlet being a potential write request (refer to Section 4.1) if it is the first request issued by a client — this results in a session being created as a side-effect that is not visible from the application.

Fortunately, the client glue is in a position to identify such requests as the client glue intercepts all requests and responses from and to the client, respectively. It is possible to keep track of all session identifiers used by the client. If the client uses a session identifier that the glue has not seen before, the glue issues it as a regular request through the UpRight library. If the client uses a existing session identifier, then the glue knows that this request will have no side effects and can safely issue it through the read-only interface. While we have implemented this optimization, it is of limited utility for the existing implementation of the BFT web application — there is a very small portion of requests that can actually be issued through this interface. We provide more details on this in Chapter 8.

6.3 Summary

In this section we have examined the design trade-offs associated with the implementation of the client glue. We identify two approaches - the proxy server approach and client module approach and analyse the benefits and

drawbacks of each approach. We choose the the proxy server approach to implement the client glue because of its applicability to virtually any web client without extra modification. We describe how we use the UpRight library in this approach. The browser connects to the Java application as a proxy HTTP server, which then forwards the request to the UpRight client library. The HTTP response returned through the UpRight client library is returned to the browser via the same TCP connection that the browser used to make the request. Thus the browser does not require any modifications, giving us the advantage that any browser can be used with the BFT web application. We also discuss the addition of the read-only request optimization provided by the UpRight library and what this means for the implementation of the client glue.

Chapter 7

Server Side Adaptation

This chapter details the modifications made to the Tomcat servlet container and the VQWiki web application in order to adapt them to suit the requirements of the UpRight BFT library. It highlights the basic design consideration when implementing communication between the UpRight server library and Tomcat, and also details the modifications made to Tomcat and VQWiki application in order to support the server library.

7.1 Design choice - Proxy client vs Server module

This section presents a design consideration for the server glue complementary to the one in Chapter 6 for the client glue. The server glue may be implemented either as a proxy client or as a module in Tomcat itself. Each has its respective advantages and disadvantages as discussed below.

7.1.1 Glue as Proxy Client

In this approach, the server glue acts as a proxy client, issuing HTTP to Tomcat requests on behalf of all clients. This is similar to the proxy server approach for the implementation of the client glue as discussed in Chapter 6.

This has a couple of advantages — the glue implementation is less intrusive to the Tomcat codebase since there is no need to modify communication code in Tomcat; it also provides an easy way to issue requests in the order provided by the Order node without worrying about the multi-threaded nature of Tomcat. We initially adopted this approach believing it to involve less complexity, but discovered that this approach suffers from some serious design flaws.

Firstly, this approach is inefficient — it requires a large amount of socket communication between the server glue and Tomcat, and adds extra per-request latency from the round trip time between the glue and Tomcat. Further, this approach complicates checkpointing and recovery as it necessitates the creation of another protocol between the glue and Tomcat in order to checkpoint and load application state. Thus, contrary to our first impression, this approach is actually more complex to implement.

7.1.2 Glue as a server module

In this approach, the glue is implemented within the Tomcat codebase, replacing Tomcat's communication and replication code. This approach is shown in Figure 7.1, which also depicts the use of a proxy server client glue. By making the server glue a modular part of Tomcat, many of the problems of the proxy client approach are avoided — the performance bottleneck is removed, as are additional points of failure, and no custom protocol is required for checkpointing and recovery of application state. However, this approach does require intrusive modification of the application server in order to replace

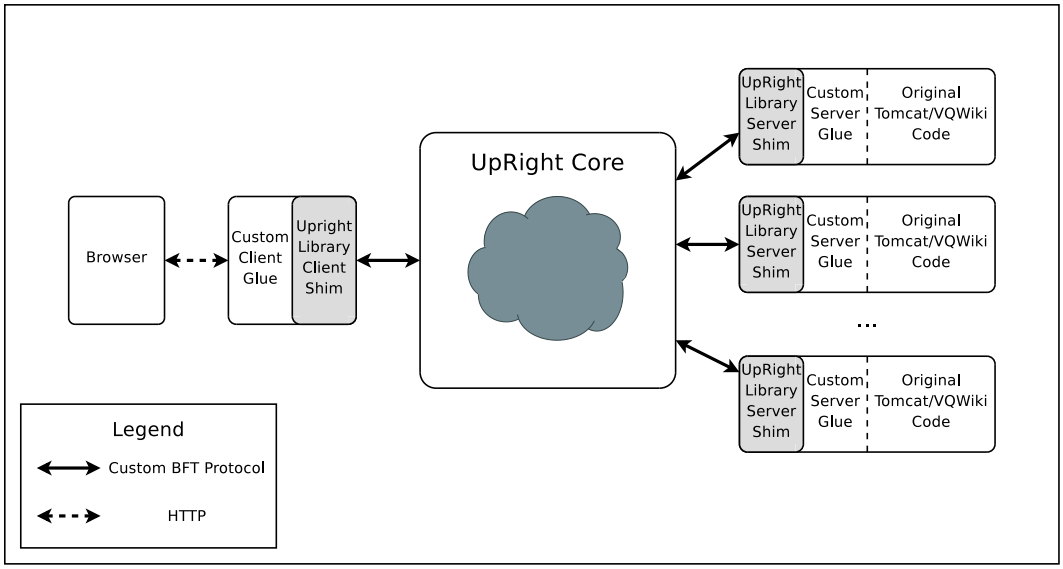

Figure 7.1: The server glue as a server Module. It is implemented by modifying the codebase of the Tomcat server and the VQWiki application

the code that handles the communication of requests and responses from and to the client respectively. In the case of Tomcat, this means replacing all the code that handles sockets and the associated data transfer along those sockets. In retrospect, however, this is effort well spent as the modifications result in a BFT Tomcat that can then serve as a building block for simpler development of web applications on top of it. We choose this approach to implement the BFT Tomcat and VQWiki application.

7.2 Implementing The Server Glue

We implement the server glue as a module integrated into Tomcat by replacing the component in Tomcat that accepts connections - the Coyote

HTTP connector [34]. This module is modified so that the incoming request is received from the UpRight library and the glue code, instead of from a TCP socket. Correspondingly, the HTTP response is sent to the client via the server glue code and the UpRight library.

We implement the hybrid checkpoint/delta approach [7] from scratch since it is not present in Tomcat. As a result, the server glue contains some extra implementation that would not be required in a typical fault tolerant application that generates its own application snapshot [7]. Functionally, the server glue implementation can be divided into request execution and recovery.

7.2.1 Request Execution

For performance reasons, the UpRight library processes client requests and issues them to the application server in batches instead of individual requests. Request batches are issued through the GlueShimInterface (`GlueShimInterface.exec()`). It is the responsibility of the server glue to issue these requests individually to the application server and in the same order across all execution replicas.

The glue passes on incoming requests to the protocol handler (in this case the `HTTP11Protocol` handler), which then processes the requests in the same way as the unmodified version of Tomcat. The response is returned to the server glue by the handler, and then sent back to the client through the `ServerShimInterface.result()` interface exposed to the server glue by the UpRight library.

For ease of implementation, we implement application snapshots through special requests that are processed through the same `HTTP11Protocol` handler interface. As a result, they are processed by the web application, which then generates application snapshots in a stop and copy manner [7]. This approach has the advantage of a simpler implementation, but now leaves the responsibility for taking and loading application snapshots entirely to the web application hosted on Tomcat. Thus presently, the application has to not only save its own state, but also that of Tomcat's (session information). For cleaner separation of functionality, the glue can include separate coordinated mechanisms for checkpoint Tomcat's session state and the application state separately. Our current implementation does not use this approach, but given the modularity Tomcat's codebase, it is not difficult for future work to separate this functionality and provide a simpler checkpointing interface to the web application.

Our implementation of the stop-and-copy application snapshot approach suffers from scalability issues on a regular file system. Typical small scale wiki deployments will be in the order of tens of Megabytes but larger wikis may occupy several Gigabytes [39]. If the expected size of the wiki is large, then the application snapshot solution needs to scale to handle this case effectively. A promising alternative to stop-and-copy is to use file-system copy-on-write support to generate application snapshots [41, 43] and use on-demand state fetching during recovery [41]. We have not explored this approach in this thesis, but it has shown to provide provide much better performance than

stop-and-copy for applications with large file-system state [41], making it an interesting avenue for future work.

7.2.1.1 Taking Checkpoints

We use the hybrid checkpoint/delta approach [7], by which checkpoints taken by the application can be divided into fine-grained deltas and coarse-grained application snapshots. Fine-grained checkpoints are taken at frequent intervals and are required to be relatively inexpensive and preferably incremental. Application snapshots are a snapshot of the entire working state of the application that are taken at relatively infrequent intervals and their purpose is to enable garbage collection of fine-grained snapshots.

Fine-Grained checkpoints - Request Logs

In the Tomcat server glue, we implement fine-grained checkpoints as incremental logs of the requests executed between successive checkpoints. As shown in in Figure 7.2, these logs are asynchronously flushed to disk while allowing the server to continue executing requests issued to it. Effective recovery using this scheme requires a replay of the logs in sequence, starting from the first log. While these checkpoints are inexpensive to generate, recovery requires a complete re-execution of every request executed from the start of the log. Since this is not a long-term solution, we use application snapshots in order to avoid replaying an arbitrary large number of logs in order to recover an execution server.

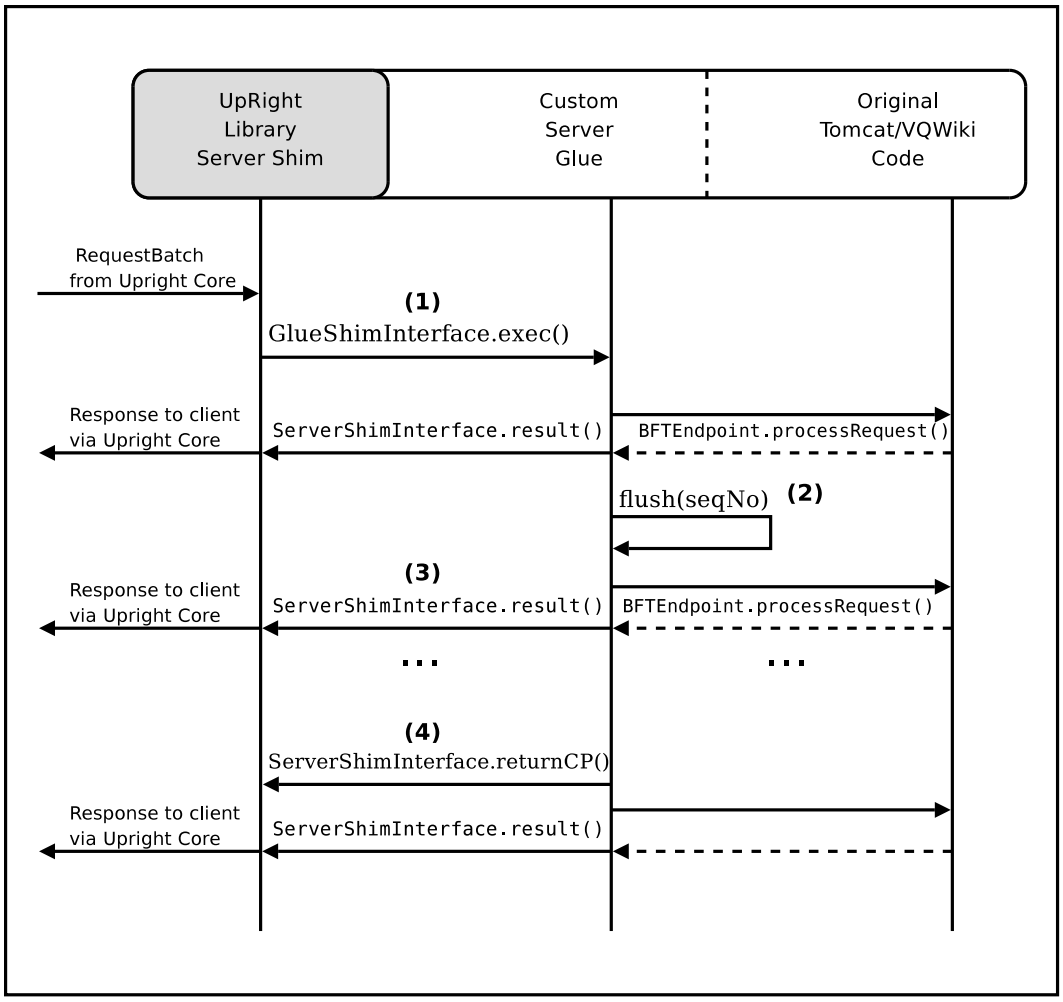

Figure 7.2: Sequence diagram for delta checkpointing in server glue.
(1) The Upright code calls `GlueShimInterface.exec()` in the server glue.
(2) The glue code flushes the logs corresponding to the sequence number of the batch currently being executed before the checkpoint is taken.
(3) The last request of the batch is executed by the application.
(4) The completed checkpoint token consisting of a log of requests from the previous delta checkpoint to the current one is returned to the UpRight library.

Application Snapshots

The Tomcat server, by default, does not provide any mechanism for taking an application checkpoint. It stores and manages no application state other than session information corresponding to the clients that connect to the web applications hosted on it. Web applications use and modify this state according to their requirements [9].

An application snapshot of a web application would require both the collection of all current sessions and their associated information, as well as a snapshot of the working state of the application. For the VQWiki web application, the working state used is a collection of files on the file-system. In practice, a database store can also be used, albeit with modifications for taking and loading checkpoints as required by the UpRight library. This is discussed further in Chapter 9.

In the Tomcat and VQWiki application, we use the stop-and-copy approach [7] to create application snapshots. We divide the process of taking the snapshot between the VQWiki application and the server glue. The process is described in Figure 7.3. First a custom HTTP request is issued by the glue to the VQWiki application (step 1). This request is processed by a special servlet (`CheckpointServlet`) added to the VQWiki application that saves both the Tomcat session state and the VQWiki application state (step 2) in a special location on the local file-system. Once the request processing is complete, Tomcat can then continue to process normal requests while the server glue completes the generation of the application checkpoint token (step 3) and re-

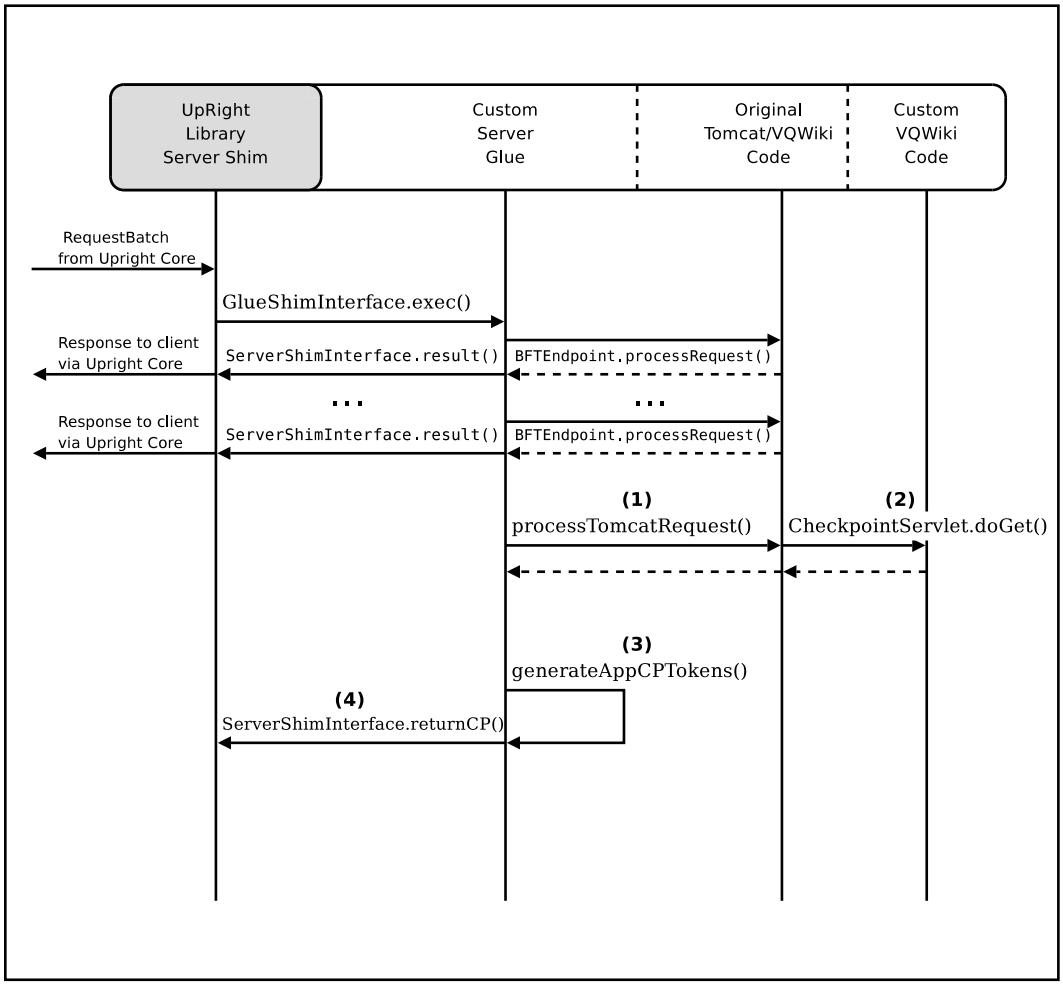

Figure 7.3: Request sequence when taking an application snapshot:
(1) A special request is issued to the `CheckpointServlet` in order to take the application snapshot.
(2) The servlet code copies the application state to a specified location on the file-system for the glue to process.
(3) The glue code generates the application checkpoint token from the state stored on the file-system.
(4) The glue returns the checkpoint token to the shim.

turns it to the shim (step 4) via the `ServerShimInterface.returnState()` call of the UpRight library. The return of the checkpoint token to the UpRight server shim marks the completion of the application snapshot with respect to this execution server. The glue divides the application snapshot into logical pieces (in this case individual files that make up the state), in order to better handle state transfer as mentioned in Section 7.2.2.2.

7.2.1.2 Releasing a checkpoint

The server glue implements garbage collection of old checkpoints through the implementation of the `GlueShimInterface.releaseCP()` call. This garbage collection is applied only to those checkpoints before the latest application snapshot. Delta checkpoints taken after the latest application snapshot need to be retained until the next application snapshot is taken, after which they can be released. This is because a node recovering to a particular delta checkpoint requires the preceding application snapshot and all deltas up to the one being loaded, in order to successfully recover its state.

7.2.2 Recovery

In the UpRight library, autonomous recovery is implemented through the transfer of application state from one execution replica to another, through the interface exposed by the library. The library thus requires that the server glue to be able to (1) load a given checkpoint based on the state provided by the server shim and to (2) fetch checkpointed state requested by the shim,

in order to transfer it to another execution replica that is recovering. This functionality is explained in the following two sections.

7.2.2.1 Loading a checkpoint

The server glue must be able to load a given checkpoint in order to recover as directed by the server shim. The glue may be requested to load a checkpoint when it falls behind in executing requests and the rest of the system has already moved to a later checkpoint, or when it has crashed thereby losing all of its volatile state. In either case, the UpRight library makes no distinction between these two types of situations and treats them the same. In order to process checkpoints more efficiently, the UpRight library operates on the checkpoint token rather than the entire checkpoint itself [7]. Consequently, when the server shim requests the glue to load a checkpoint, it provides only the checkpoint token. It is the responsibility of the glue to process this token and obtain the required application state, either from its own local disk or from the transfer of state from another execution replica that has the required state. For performance reasons, the application state is divided into logical pieces, each with its own token. The glue processes the collection of state tokens in the checkpoint token and requests the shim for the respective pieces through the `ServerShimInterface.requestState()` call.

The sequence of loading a checkpoint is described in Figure 7.4. The server shim issues a call (`GlueShimInterface.loadCP()`) to the glue requesting it to load a particular checkpoint by providing the corresponding check-

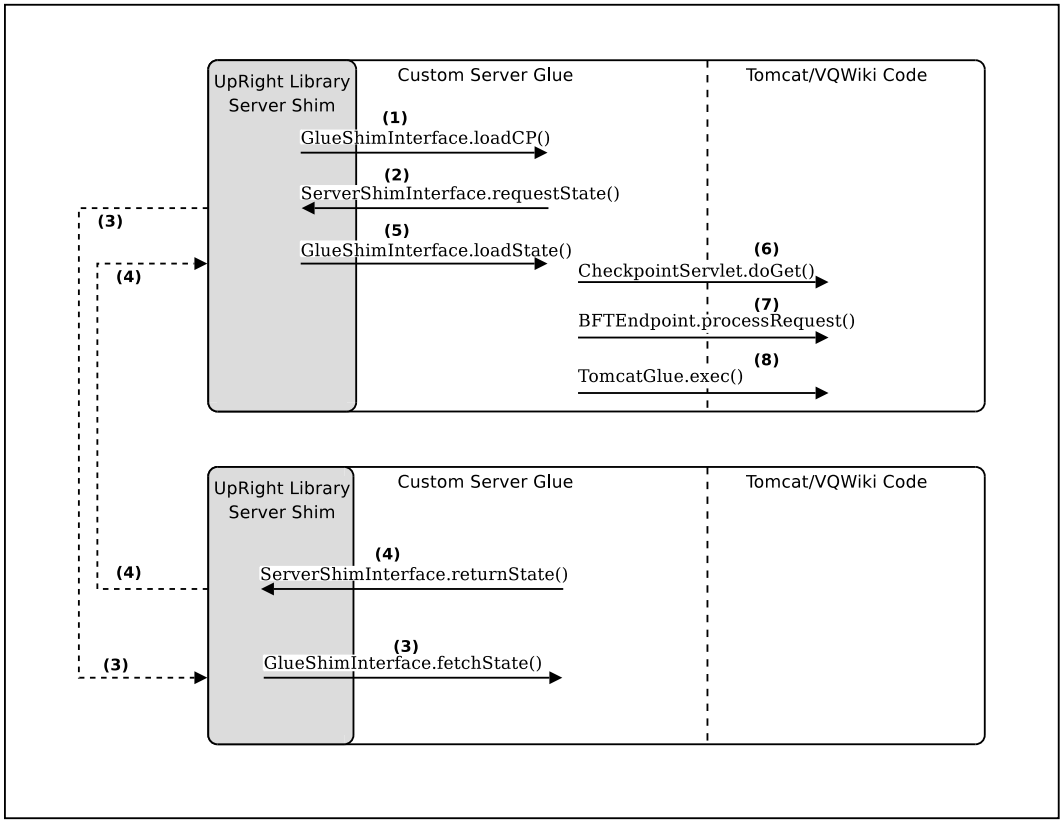

Figure 7.4: Sequence of requests in loading a checkpoint:
(1) The glue receives a request from the shim to load a checkpoint, given a checkpoint token.
(2) The glue requests the shim for all the pieces of state specified in the checkpoint token.
(3) The shim requests another shim for the state that the glue requested. The other shim requests its own glue to fetch this state.
(4) The other shim returns the requested state.
(5) The shim returns the state to the glue.
(6) When the entire application snapshot is available, it is loaded into the application.
(7) If there are any delta logs, these are replayed.
(8) If there were any requests received while the snapshot was being loaded, these are processed in the order they were received.

point token (Step 1). The glue then processes this token and asks the shim for pieces of the checkpoint through the `ServerShimInterface.requestState()` call to the server shim (Step 2). The server shim then issues requests to another server shim to fetch these pieces (Steps 3 & 4). Once the state is transferred back to the requesting server shim, it is returned to the server glue via the `GlueShimInterface.loadState()` call. The glue in turn loads the state in place (Step 5). Once all the required pieces of state are in place, the server glue then loads the checkpoint (Step 6 & 7), which may consist of loading an application snapshot (Step 6) and possibly replaying request logs (Step 7) in order to bring the application to the state specified by the checkpoint. As a further optimization, the server glue does not stop the server shim from issuing requests to it while it is loading a checkpoint. Instead it caches these requests and continues executing them once the checkpoint is completely loaded (Step 8). This optimization allows the system to progress and prevents the stalling of the UpRight core while the recovering Execution node is catching up [7].

7.2.2.2 Fetching Application State

The UpRight server glue interface requires that, given a token of a particular checkpoint, the server glue be able to fetch application state corresponding to that checkpoint and return it to the shim. This call (`GlueShimInterface.fetchState()`) from the library occurs as a direct consequence of some execution replica making a request to its server shim for that corresponding state (through the `ServerShimInterface.requestState()` call).

This functionality enables the transfer of checkpointed server state from one server replica to another in order to load a checkpoint, effectively allowing the recipient replica to recover autonomously [7]. In the Tomcat glue, we implement the fetching of application state by a background processing thread.

Since the server glue divides the application state into logical sub-pieces, the glue receives requests to fetch individual pieces of the application snapshot, rather than the snapshot as a whole. This gives the underlying library a chance to optimize on state transfer by requesting different sub-pieces of state from different server replicas, thereby conducting state transfer in parallel.

7.3 Modifications to Tomcat

In addition to the implementation of the server glue, Tomcat and VQWiki required some invasive modification in order to conform to the requirements of the UpRight library. In this section, we describe these modifications in more detail.

As mentioned in Chapter 4, the UpRight library has certain theoretical requirements from the application it is to support. These arise from using the replicated state machine approach used by the UpRight library [7, 31]. While we apply these for the purpose of making Tomcat Byzantine fault tolerant, the principles apply equally to any fault tolerance method using the state machine replication approach.

7.3.1 Removal of non-determinism

One of the most significant requirements of the UpRight library with respect to modifying Tomcat is the removal of non-determinism in request execution. In Tomcat we identify three sources of non-determinism — multi-threaded execution, randomness and dependence on real time.

7.3.1.1 Making Tomcat Single-Threaded

By default, Tomcat is multi-threaded in order to improve scalability with respect to the number of clients it can simultaneously handle. However, the UpRight library requires that all replicas execute requests in the same order which cannot be enforced within a multi-threaded server without making thread scheduling deterministic. While this has been done in previous work for a Primary-Backup architecture [3, 29], it is beyond the scope of the present work. Another option beyond the scope of our work is issuing independent requests in parallel [22, 36].

Instead of modifying the thread scheduler or implementing a sophisticated scheme to identify independent requests, we enforce the same sequence of request execution across all Tomcat replicas by executing all requests in serial order using a single request processing thread. This has performance implications which we observe in our evaluation and we propose ways of improving on this (refer to Section 8.2.4).

7.3.1.2 Removal of Randomness

Another source of non-determinism is the use of randomness within Tomcat to generate identifiers such as the JSP session identifier [9]. In order to generate secure session ids, Tomcat uses all possible sources of randomness from the underlying operating system. Since the session ids need to be the same across all execution replicas, this randomness needs to be removed. We replace the non-deterministic random number generator `SecureRandom` with the deterministic random number generator `java.util.Random` and seed it with the seed provided by the server shim as part of the `GlueShimInterface.exec()` call. While this may be a security concern from the point of view of the session id (since the session id generation is now deterministic), we argue the security provided is at least as strong as that provided by the seed generator of the UpRight library. If there is sufficient randomness in the seed provided by the UpRight library, then this randomness will be preserved by Tomcat's deterministic random number generator.

7.3.1.3 Removal of dependence on real time

In addition to randomness, real time also plays a role in the non-determinism of server execution. This is in the form of background threads and responses that depend real time.

Housekeeping threads The UpRight library mentions "housekeeping" threads as sources of non-determinism [7]. These are threads that depend on real time

to perform actions that modify the state of the server. Tomcat uses housekeeping threads to perform processing of periodic time-dependent tasks such as expiring client sessions that have been inactive (i.e., no client requests have been made for that session) for a certain period of time. Since this depends on real time provided by the server's local clock, this needs to be modified in order to use the virtual time provided by the UpRight library to enable each of the server replicas to appear perform all such maintenance tasks at the same time as observed from the clients.

Fields containing real time The HTTP protocol specifies fields in the HTTP response that include timestamps based on real time such as the `Date` and `Last-Modified` fields [12]. While there are other fields that also depend on real time, we concentrate on these two fields in the following discussion.

— The `Date` field is used to timestamp the HTTP response and depends on the system clock. In order to remove the variation in this field across replicas, we use the virtual time provided by the UpRight library instead of the system clock. Virtual time is provided by the UpRight library in the `GlueShimInterface.exec()` call and is generated from the local system clock of one of the order nodes of the UpRight library. Since the virtual time is updated before an incoming request is processed, the semantics of client session expiration (which is also based on real time) remain unchanged. Further, this has the effect from Tomcat's point of view, that all requests in a batch are executed at the same instant in

time, albeit in serial sequence because of single-threaded execution. By modifying the reference to system time across Tomcat, all such fields that depend on the system clock, now depend on virtual time provided by the UpRight library.

— The `Last-Modified` field is typically used for cache control [12] when retrieving static content such as files from the file system of the server via HTTP. In this case, the value of the `Last-Modified` field depends on the file system's last modified date of the file. Since this information cannot be controlled in the same way that we control Tomcat's view of system time (by using virtual time), we ensure that each static file has the same file system modification timestamp across all replicas, by modifying the file system attributes consistently across all replicas. While this is external to the system and needs to be performed when the system is quiescent, it is expected that updates to static content will be infrequent enough to permit such an approach. If such an assumption cannot be made, then further modification of Tomcat code that handles static files is required, in order to use a custom last modified timestamp for all static content. However since this is not central to the goal of making Tomcat BFT, this has been treated as an optional feature and not been implemented. In our implementation we assume that all files have the same file system timestamp so that the `Last-Modified` field of the HTTP response is the same across all replicas.

7.3.2 Replacement of network module in Tomcat

In order to implement the server glue as a module of tomcat, we replace the networking code in Tomcat that is responsible for receiving requests and sending responses. The point chosen to make this replacement was in the Coyote connector. The connector consists of an endpoint class (`JIOEndpoint`) that accepts incoming HTTP requests on TCP sockets and passes them on to the protocol handler (`Http11ConnectionHandler`) which then continues normal processing of the request. To adapt the connector to the UpRight library, we replace the `JIOEndpoint` with a custom `BFTEndpoint` that provides the protocol handler with a fake `Socket` object containing the data of the request. Since the socket is not a real socket, this raises the issue of accessing the client host name, IP address and remote port through the socket, as is presently done through the Servlet API [9]. We propose that by using the client Id provided by the UpRight library along with the request, the required client information can be obtained in a lookup table, should it be needed by the application. While this feature has not been implemented, it is straightforward to implement if desired.

7.3.3 Application snapshot - client session management

Tomcat possesses no mechanism for taking or loading an application snapshot. It does however possess utility methods (in the `StandardManager` class) for persisting, loading and modifying the list of client sessions stored in main memory, which make up Tomcat's working state. We extend these

utility methods (through the `BFTStandardManager` class) in combination with a special servlet (`CheckpointServlet`) added to the VQWiki application in order to persist and load the client sessions as part of an application snapshot.

7.4 Adaptation of the VQWiki Web Application

In order to adapt it to the requirements of the UpRight library, the VQWiki web application also requires some modifications, specifically the removal of non-determinism and implementation of snapshot handling.

7.4.1 Removal of non-determinism

Similar to the removal of non-determinism in Tomcat, we remove dependence on real time and non-deterministic randomness.

7.4.1.1 File-system dependence

The VQWiki application uses the file-system to implement coarse-grained locking for coordination between different users attempting to modify a particular wiki topic. In the original VQWiki application, lock timeouts are implemented by comparing the last modified time of the lock file to the current system time. We replace the current system time with the virtual time provided by the server shim in the `GlueShimInterface.exec()` call. To implement lock timeouts, instead of using the last modified time from the file system, we store the virtual time corresponding to the lock file's last modification, as data within the file. This way the lock will appear to be created at

the same virtual time across all replicas. It will also expire at the same virtual time across all replicas.

7.4.1.2 Admin password generation

The VQWiki application restricts access to certain content and settings by making them available to only an authenticated admin user. By default, the VQWiki application generates a random admin password in a non-deterministic way. Since this would produce different results across replicas, we disable this password generation feature. Instead, we require that the password must be set to a desired value on all the replicas before starting the application. We implement this by setting a default value for the password at application start-up, and having the user change it through the UpRight library. Also, the admin password can be modified subsequently at any time by accessing the web application through the UpRight library.

7.4.1.3 Temporary folder for uploaded files

In order to handle uploaded files as attachments to wiki pages, the VQWiki application uses a temporary folder to store the uploaded files, before storing them in their final storage location. While the final stored file data does not depend on real time, the temporary location uses the servlet upload API that generates filenames based on real time. Since the temporary folder is not used for the working state of the application, we exclude it from the state that comprises the application snapshot.

7.4.2 Checkpoint Servlet

Our approach to implement the generation and loading of application snapshot capability in Tomcat requires the addition of an extra servlet (`CheckpointServlet`) to the VQWiki application. Application snapshot generation is done in two stages as described in Section 7.2.1.1. The `CheckpointServlet` servlet copies all the application state into a special location on the file system for the glue to generate tokens from. This application state consists of the sessions stored in Tomcat's session manager (`StandardManager`), and the VQWiki store which contains all the content of the wiki, including attachments, settings and lock files. The sessions are saved through a customized extension of the `StandardManager`, which provides an interface to serialize and de-serialize all currently active client session information, while the VQWiki store is saved by copying the relevant files into the special file-system location managed by the server glue. The checkpoint servlet is responsible for calling the session manager in order to save the session information as part of the application checkpoint state. While this leaves more responsibility in the hands of the web application developer, this approach results in a cleaner implementation that meshes well with Tomcat's structure and is simple to implement. Future extensions to this work can use a special servlet outside the web application and within Tomcat's codebase to save client sessions for cleaner separation of functionality between Tomcat and the web application.

7.4.3 Summary

In this chapter we have described the implementation of the server side of the BFT Tomcat and VQWiki application. We implement the glue as a server module, and handle request execution and checkpointing through custom code written as an adapter between the UpRight server shim and the Tomcat request processing code. We implement the hybrid checkpoint/delta approach in order to generate checkpoints, and use the stop-and-copy method of generating application snapshots. In order to meet the requirements of the UpRight library with respect to state machine replication, we remove the non-determinism present in the Tomcat server. We also remove non-determinism present in the VQWiki application and add functionality in the application to take application snapshots.

Chapter 8

Evaluation of Methodology

In this chapter we present the evaluation of our implementation of the BFT Tomcat and VQWiki application. We evaluate our approach based on the ease of implementation of the system and microbenchmark performance compared to the original system.

8.1 Engineering effort

One of the aims of this thesis is to make it possible to add BFT to web applications with modest engineering effort. We use the lines of code (LOC) metric to evaluate the ease with which the system was implemented. On the server side, we evaluate the modifications made to Tomcat separately from those made to the VQWiki application because, we believe that the modifications made to Tomcat can be re-used for any web application hosted on it, thereby bringing down the cost of implementing multiple BFT web applications on the Tomcat server. Unless otherwise noted with an asterisk (*), all lines of code are in Java

Component	Lines of code
Original Tomcat code	320,167 lines
Newly-added code	3,273 lines
Existing code modified for virtual time	248 lines
Removing nondeterministic randomness	25 lines
Unused communication, replication code	29,052 lines
XML configuration*	5-10 lines

Table 8.1: Modifications made to Apache Tomcat (in lines of code)

8.1.1 Modifications to Tomcat

Table 8.1 summarises the modifications made to Apache Tomcat. Approximately 1% of extra code is added as glue code. The bulk of this code (1,241 LOC) was written in order to handle the hybrid checkpoint/delta approach [7], while some code (345 LOC) was used to handle communication between Tomcat and the UpRight library. We found that the modularity of Tomcat's codebase helped to keep the adapter code (345 LOC) small in size. We handled client sessions in a deterministic way and added these sessions to the application snapshot (1,019 LOC). The remainder of the code was in utility classes for the above functionality.

In order to remove all sources of non-determinism from the Tomcat server, we modified 248 LOC to use virtual time provided by the UpRight library [7] instead of real time. We modified all random number generators (25 LOC) to use seed values provided by the UpRight library [7] instead of utilising file system randomness and real time for pseudo-random number generation. Both these were performed by modifying all occurrences of `System.current-`

`TimeMillis()`, `java.util.Random` and `java.util.Date`, all of which required simple text searches within the source files.

By replacing Tomcat's communication and replication code, 29,052 lines of code were no longer used. This includes the cluster deployment and group messaging protocol code (28,252 LOC), both of which were replaced by the UpRight library code. It also included 800 LOC of networking I/O code used by the HTTP connector. We do not account for the removal of multi-threaded request processing since this happens as a side-effect of replacing the networking code, which controls the number of threads that process requests in parallel.

In order to make modifications to the Coyote connector (refer to Chapter 7), the `server.xml` configuration file required minor modification to configure it with property files used by the UpRight library. This is the only modification made that was not done in Java.

8.1.2 Modifications to VQWiki

Table 8.2 summarises the modifications made to the VQWiki application. We added code to checkpoint application state (551 LOC), which was not present in the original application. 66 LOC had to be modified to use virtual time for timestamps, version file names and lock expiration (refer to Chapter 7).

The VQWiki application used file system attributes for its lock files. The structure of the lock file had to be modified (33 LOC) in order to remove

Component	Lines of code
Original VQWiki code	36036 lines
Newly-added code	551 lines
Existing code modified for virtual time	66 lines
Removing file-system non-determinism	33 lines
Environment configuration*	20 lines

Table 8.2: Modifications made to VQWiki (in lines of code)

the dependence on the lock file's last modified attribute. Finally, basic environment configuration of the VQWiki application was done in order to modify the default storage of working state and transient storage in order to cleanly separate them for the purpose of generating deterministic checkpoints. This was done in both Java code as well as property files and is thus annotated with an asterisk (*) in Table 8.2.

8.1.3 Implementation of the client glue

We implemented the proxy server client glue in 814 lines of code (refer to Chapter 6). This functionality includes, parsing the client request, forwarding the request to the UpRight library, and returning the response sent by the replicated server through the UpRight library, back to the client.

8.1.4 Summary

From the above evaluation it is clear that the addition of BFT to the Tomcat and VQWiki applications is greatly simplified by using the UpRight library. This avoids having to deal with the replication and fault tolerance pro-

tocols because they are handled by the library code (which, according to [7], amounts to 20,403 LOC). While Apache Tomcat required some extensive modifications, we expect that this cost can be amortized over the implementation of multiple BFT web applications that all reuse the framework provided by Tomcat.

While it took 4 months for a single student to implement this system, it must be noted that the process of implementation was evolutionary, and hence many of the design alternatives, including the ones that were discarded, were evaluated in the course of the actual implementation. We expect that future implementations would be able to reduce development time by taking advantage of the lessons we have learnt in this implementation and directly using the best practices identified by this thesis.

8.2 Performance Evaluation

We tested the performance of the original and BFT versions of the Tomcat and VQWiki applications on 2.6GHz quad-core Xeon machines for the wiki application servers, and 3GHz dual-core Pentium-IV machines for the order and RQ machines required by the UpRight library [7]. All nodes use the Linux 2.6 kernel and the Sun Java 1.6 JVM. Unless otherwise noted, separate machines were used for each of the order, RQ and application servers. The nodes are connected by a 1Gbps Ethernet in order to test the round-trip latency of client requests without including network delays and packet losses. We used VQWiki 2.8.1 with a file-system data store and an unreplicated Apache

Tomcat 6.0.18 server in its default configuration for our baseline implementation. For the BFT version of the server we used *hybrid checkpoint/delta* checkpointing with *stop-and-copy* [7] generation of application snapshots and request logging for delta checkpoints in the server glue. We configured the system to handle one crash or one Byzantine fault ($u = 1/r = 1$) [7].

Instead of using web browsers as clients, we use Java client programs. In the original wiki application test, clients use the Apache HTTPClient library [17] for client-server communication, while the test clients for the BFT wiki application use the UpRight client library [7]. In order to model identical client behaviour in both the systems, both types of clients issue only one request at a time, with no think time between requests. While this is not an expected per-client workload, it enables us to effectively stress test the system with fewer clients.

For our experiments, we used a microbenchmark workload of topic read and topic write requests to measure the performance of the two versions of the server. The requests directly accessed a wiki topic that was uniformly chosen at random.

We note that while the size of the request was fixed for a certain workload, the size of the response varied from a few hundred bytes to tens of kilobytes. This is because for write requests, the server issues a `HTTP 302 - Moved Temporarily` response [12] which is always the same size, and for read requests, the response included a list of pages that were previously accessed by that client, which increased with the number of unique pages previously

fetched by the client.

For simplicity, we did not include requests for static content such as images or text files because in a real browser environment that implements cache control, these images would be fetched only once and then stored in the browsers cache. Subsequent requests would never be made to the server until the cached copy expires. Hence our workload consists entirely of requests that are processed by servlets in the VQWiki application, in order to accurately measure processing overhead. We find that despite its simplicity, this approach highlights some interesting issues with the VQWiki application which are further discussed in Section 8.2.3.

Fetching images brings up another issue we faced — our analysis showed that image fetches were the only truly read-only requests with respect to our application. Every other request to the VQWiki web application involved the modification of the client session information as a side effect of executing the request. We thus use a write-only workload with respect to the UpRight library (we do not use the read-only optimization [7]). We find that on its own, this decision does not sacrifice performance since we observe reduced throughput due to the serialization of requests in the Tomcat glue (see Section 8.2.2). However, this optimization in conjunction with multi-threading of read request execution at the server should provide some interesting improved results and we plan to explore this possibility in our future work.

Our stop-and-copy approach to generating application snapshots, has obvious performance overheads [41]. We therefore measure the performance

of our BFT web application using only delta checkpoints by request logging. Future work can evaluate the use of better application snapshot approaches such as copy-on-write [41, 43], but these are beyond the scope of this thesis.

8.2.1 Throughput Comparison

We conducted experiments to measure the throughput and latency of the original and unmodified versions of the system for a read-only workload, a read-write workload with a 90:10 ratio of reads to writes, and a write-only workload. It must be noted that the reads and writes referred to here are those of wiki topics and not requests in the UpRight system. All our client requests pass through ordering in the UpRight system as we do not use the read-only optimization in our measurements for reasons specified in the preceding section.

8.2.1.1 Read-only workload

Figure 8.1 shows the results of our microbenchmark performance testing for a read-only workload of 1KB requests. We believe that the overhead of the BFT version of our web application lies in the single-threaded execution of requests. Our results from running the original version of Tomcat as a single-threaded server confirm this hypothesis (see Figure 8.7). Because in our BFT version all requests are issued serially to Tomcat by the server glue, the VQWiki web application is unable to take advantage of any parallelism that the unmodified Tomcat server provides. In Section 8.2.4.1 we explain how multiple

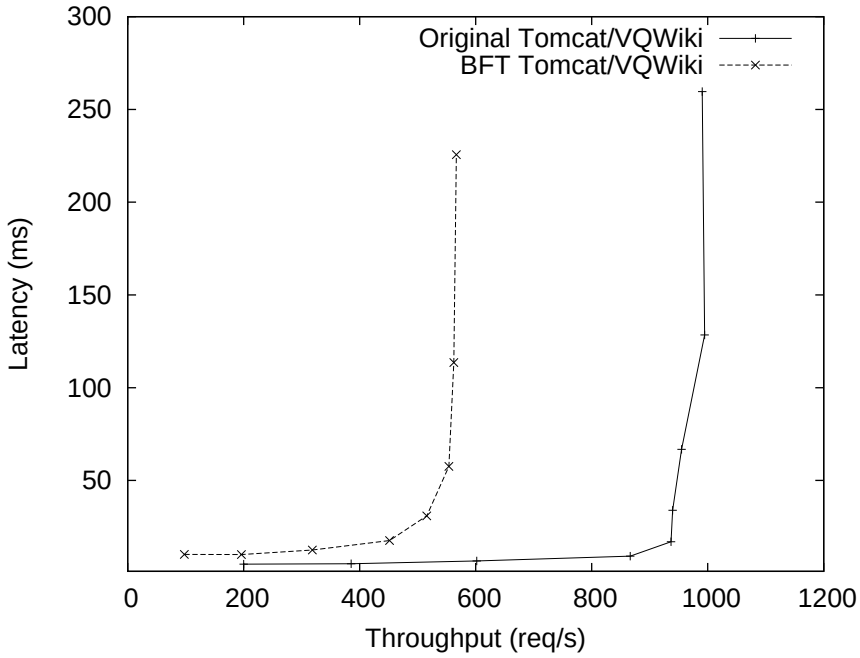

Figure 8.1: Response time vs throughput for a 1K read-only workload - comparison of the original and BFT wiki application

threads can be used to execute reads thereby improving the performance of the BFT implementation. We were, however, unable to evaluate this using a realistic workload for reasons specified in Section 8.2.4.1.

8.2.1.2 Read/write workload

We ran a read/write 1k workload consisting of 90% reads and 10% writes, the results of which are presented in Figure 8.2. However we observed

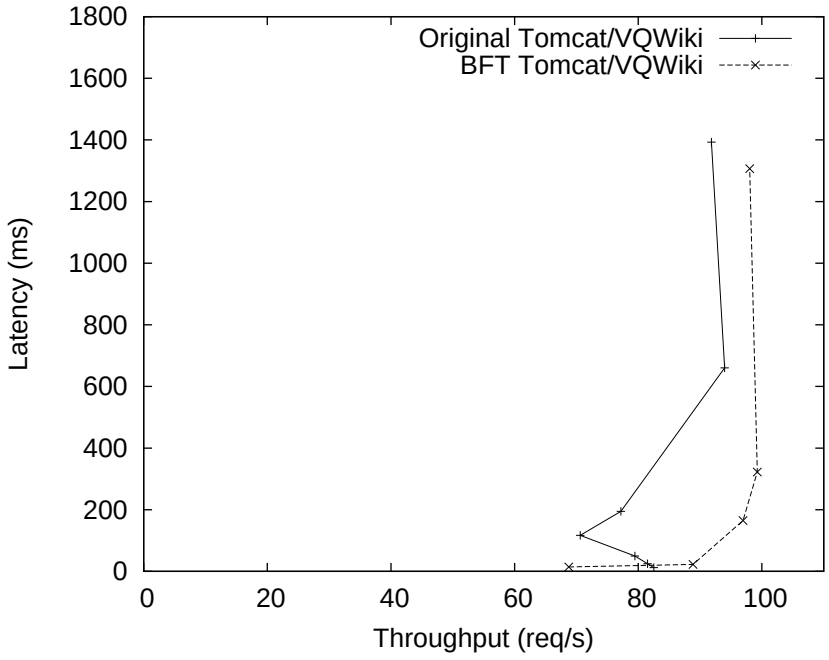

Figure 8.2: Response time vs throughput for a 1K 90/10 read/write workload - comparison of the original and BFT wiki application

an anomaly in the throughput of the original application which is not present in our BFT application (we do not present these results here). We also found that our performance is surprisingly better. We believed that this might be due to read-write conflicts occuring in the original web application due to multi-threading, and so we attempted to verify this by code inspection and further experiments.

On inspecting the code of the VQWiki application we found that the

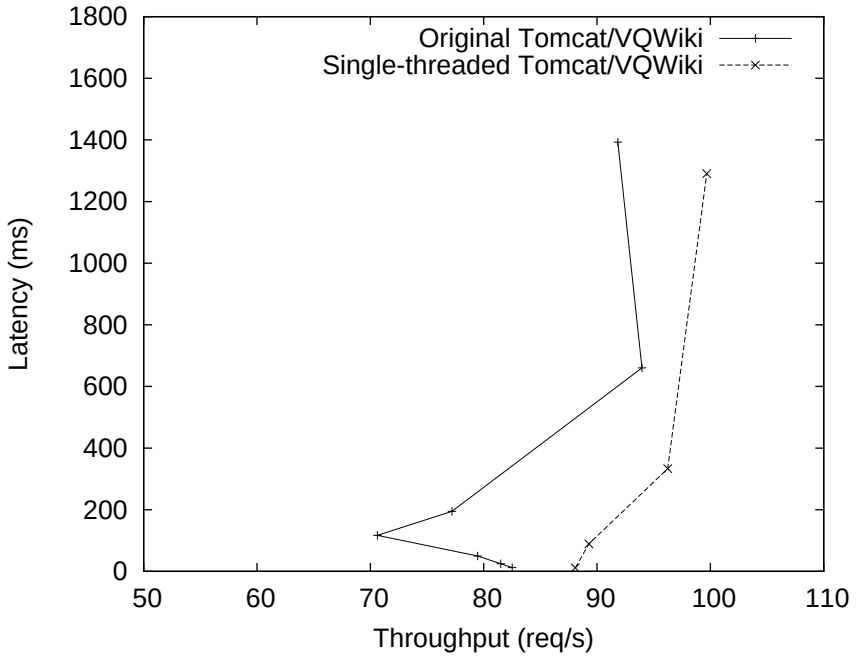

Figure 8.3: Response time vs throughput for a 1K 90/10 read/write workload on the original and single-threaded Tomcat/VQWiki application

Apache Lucene [24] search index used by the application, is accessed for every read and write, leading to possible contention within the Lucene code. Since VQWiki uses an old version of Lucene [37], the source code was unavailable for inspection in order to identify possible bottleneck areas in the Lucene code.

Without access to the source of the Lucene search engine, we turned our attention to the workload. In our original experiments clients would write identical files containing the same random sequence of characters. We assumed

that this might be causing a skewed distribution of data within the search index code, possibly making it inefficient.

In order to test this theory, we then modified this client behaviour to generate random dictionary words that were unique across files and across multiple requests, and thus the results in Figure 8.2 depict those updated experiments. We find that the original system still shows a drop in throughput at medium load.

Finally, in order to verify whether this behaviour was actually caused due to multi-threaded read-write conflicts, we ran the original Tomcat server with a single request-processing thread in order to see if the anomaly reproduced itself. We find, as shown in Figure 8.3, that it did not happen for a single-threaded original server. We also observe that the performance of the single-threaded original server matches that of our BFT server in Figure 8.2.

Since this drop in throughput happens only with the results obtained when running the original multi-threaded server, we conclude that this is due to the synchronization of conflicting reads and writes, performed by the Lucene search index in the VQWiki application.

8.2.1.3 Write-only workload

When running a write-only workload we observed that both systems get saturated with just two clients, indicating that writes have a high latency. We profiled the server and found that a majority of the time spent in executing the write was spent in updating the Lucene search index. Surprisingly, how-

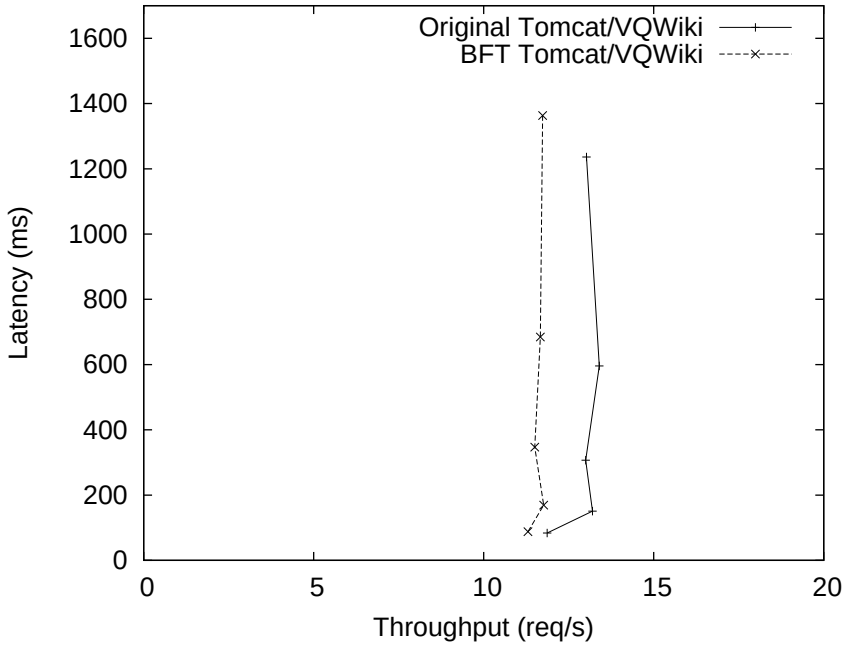

Figure 8.4: Response time vs throughput for a 1K write-only workload - comparison of the original and BFT wiki application

ever, we did not find any write-write conflict anomaly in the multi-threaded server as we found with the read-write workload. Anomalies aside, we find that the BFT application performs almost as well as the original application (see Figure 8.4). Code inspection reveals that this is because writes are serialized by the VQWiki application, thereby removing most of the performance advantage of multi-threaded request execution in Tomcat. However, we note that the small performance difference is due to the difference of the location

at which requests are serialized in the two systems. In the original system, this serialization happens at a fine-grained level within the servlet code at the functions that perform writes to the file-system, whereas in our system the serialization happens within the glue at the coarse-grained level of requests to the server. Hence the unmodified server still utilizes some parallelism from multi-threaded execution which contributes to its marginally superior performance. Again, our results in Figure 8.7 show that when requests are serialized by making the original server single-threaded, performance is the same as the BFT version, confirming this hypothesis.

8.2.1.4 Workload sensitivity

Since we observed that the system performs badly with writes, we measured the sensitivity of throughput as the ratio of reads to writes varies. From Figure 8.5 we find that as expected, the throughput of both applications dramatically reduces as the number of writes increases. The large performance difference seen in the read-only workload is caused by our serial execution of reads as opposed to their parallel execution in the original server. We propose ways of improving this performance in Section 8.2.4.

8.2.1.5 Sensitivity to request size

We ran experiments to measure the original application's sensitivity to the size of the requests issued. We varied the wiki size from 1000 files of 100 bytes each to 1000 files of 10K bytes each, resulting in a server state

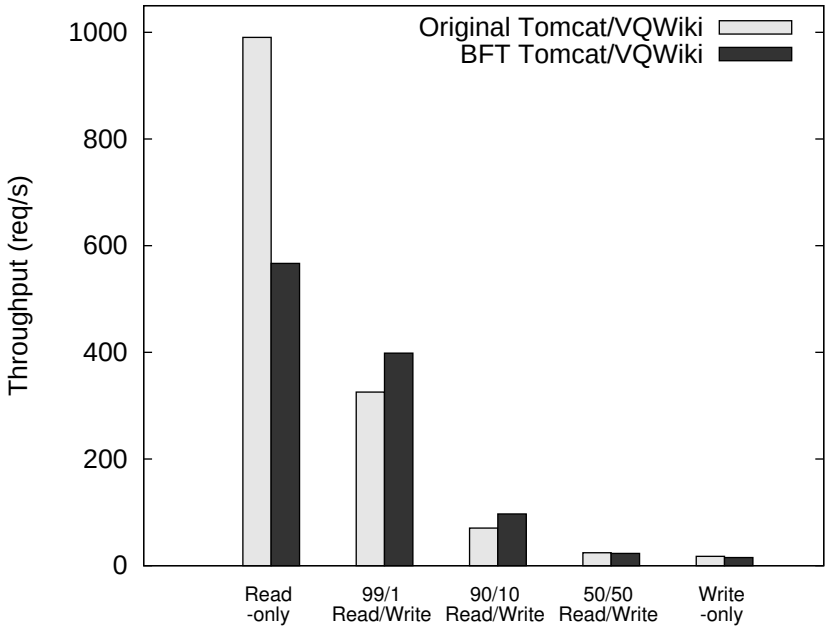

Figure 8.5: Throughput comparison of the original and BFT wiki application for different read/write mixes

that varied from approximately 1 MB to 100MB. The size of the request was matched with that of the file size in order to maintain the same amount of working state throughout the run of the experiment. We ran the write-only 1K workload to measure throughput of the two systems.

We found that increasing the request size dramatically reduces throughput as seen in Figure 8.6. We find that a large amount of execution time is spent in indexing new pages when they are written. As request size increases,

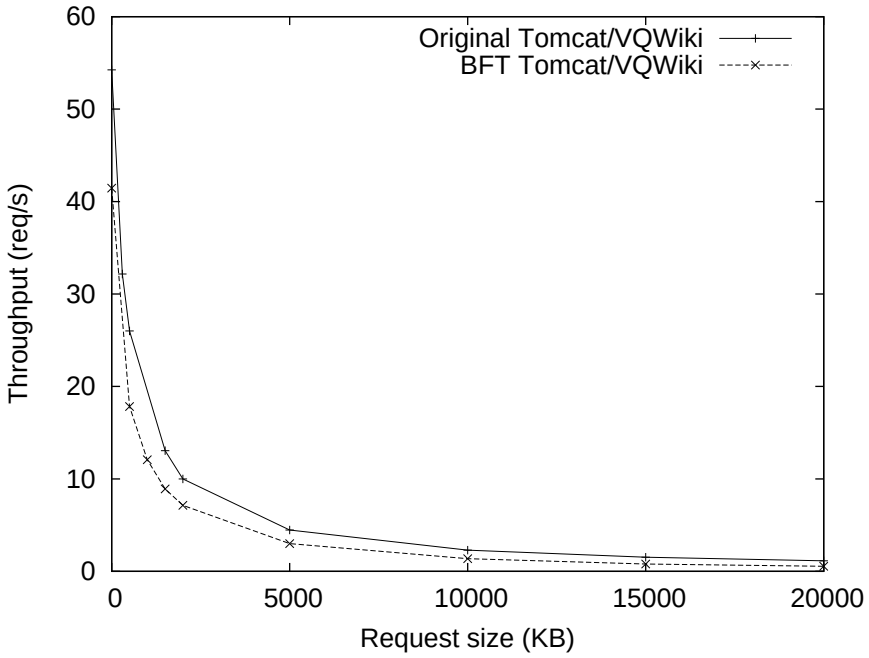

Figure 8.6: Throughput vs request size for the write-only workload - comparison of the original and BFT wiki application

so does the size of the index and consequently the amount of time spent in updating it. While the file system also contributes to this performance degradation, we found that its contribution was not as large as that of the indexing.

8.2.2 Comparing single-threaded performance

Since we observed a consistent performance degradation compared to the baseline implementation when executing read-based workloads, we decided

to evaluate the performance of the original VQWiki web application on a single-threaded Apache Tomcat server. A comparison of the results of a single-threaded Tomcat server against our BFT implementation are presented in Figure 8.7 for different read/write 1K workloads.

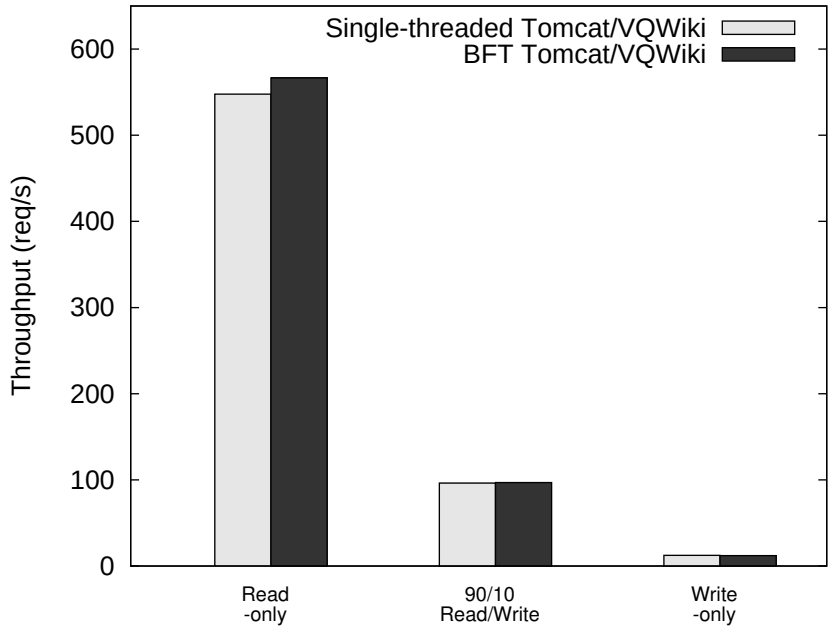

Figure 8.7: Throughput comparison of a single-threaded Tomcat server vs the BFT Tomcat server for different read/write mixes

As expected, our BFT server performs no worse than a single-threaded Tomcat server. However our performance on the read-only workload is slightly better. This is because Tomcat uses a single-threaded socket implementation

to read and write bytes to and from the network, respectively. The UpRight library [7] on the other hand uses multiple threads to read and write network bytes, giving it better performance on a multi-core system when the application is not the bottleneck. In the other workloads the application becomes the bottleneck, making the throughput more or less equal in both cases.

8.2.3 Scalability issues

We are aware that the filesystem storage mechanism of VQWiki has known scalability issues for large-sized wikis [35] but these are not elaborated. From our experiments we find that these issues are centered around maintaining an index of all the topics. In the course of our experiments we observed that the VQWiki web application did not scale well when using large-sized wikis. We find that this produces a significant overhead for write requests and identify two main reasons — topic versioning and indexing.

Topic versioning: VQWiki implements topic versioning by saving a timestamped copy of the topic being modified. This results in a file creation for every request, which we observe causes a significant performance overhead for large files. Further, this creates scalability problems when executing a large number of writes as we do in our performance testing.

Indexing: VQWiki implements a filesystem-based search index of all topics using the Apache Lucene search index [24]. However the filesystem-based VQWiki web application does not scale well with the size of the wiki being stored, due to performance issues from searching, indexing and cross-

linking of topics [35]. When using large topic file sizes and a large number of topics, we found that the application server approached system resource limits (memory usage, number of open files) with a modest workload of client requests.

According to VQWiki documentation [35], using a database for storage does not have the same issues. Recent work has demonstrated a BFT database [36] and this can be leveraged to evaluate the performance of the database versions of the original and BFT wiki applications. This remains an avenue for future work.

Another option to explore is the use of in-memory storage for the index. We have not tested this theory but we believe that an in-memory index should provide us much better index performance, allowing us to better measure the other characteristics of the system.

8.2.4 Scope for improvement

From our initial results, we found that our coarse-grained serialization of requests at the glue poses a performance bottleneck for read requests. Also, for the 90/10 read/write workload, we observed that there is a high variation in per-request latency which we attribute to read-write contention at the search index level.

8.2.4.1 Executing requests in parallel — circumventing coarse - grained serialization

From our initial results, we found that our coarse-grained serialization of requests at the glue poses a performance bottleneck for read requests. We believed that allowing multi-threaded read execution (refer to the beginning of Section 8.2) should significantly narrow the performance gap. To this end, we implemented multi-threaded execution of read-only requests in our BFT server. However it is to be noted that in our VQWiki application, the read-only optimization can be applied to a very small subset of requests — those that access static files.

In the VQWiki application the only static files available are images of very small size (< 4K) that represent buttons and other UI components on the resulting page that is fetched by a web browser. When a web client is used, these images are fetched along with the main page that forms the response. All other requests access a servlet and include the side-effect of modifying the user's session for various purposes. Even for requests that just read a wiki topic and do not modify it, client session information is modified in order to update the history of pages viewed so far by the user. The end result of this side-effect is that we have to serialize even read requests on wiki topics, in order to satisfy the requirements of our state machine replication approach. As we do this serialization in the glue, we sacrifice the parallelism that the unmodified application takes advantage of. We believe that other web applications that provide a larger percentage of side-effect free read-only requests, might allow

us to test this optimization better by executing more reads in parallel.

8.2.4.2 Faster application snapshots — file-system support

From our experience, stop-and-copy has obvious performance penalties for large data sizes. We do not present these results here, but related work has evaluated the penalty of this approach [41]. An interesting option to explore in future work is the use of file-system support to generate copy-on-write snapshots. ZFS [43] is one such file-system that provides this support, and is shown to provide performance benefits for file-system snapshot generation [41].

8.2.5 Determining usability despite scalability issues

We found that most of the scalability issues observed in VQWiki are due to assumptions made in the implementation about the nature of its expected workload. Locks are not intended to be fine-grained and updates to wiki topics are expected to be spaced out over seconds or minutes. Given that this is the case when the clients are controlled by humans, this is not an unreasonable expectation for a typical wiki deployment. Moreover, in typical web applications, 1-2 seconds can be considered the maximum acceptable latency for a request/response round trip [26].

We attempted to find out if the application is still usable despite its apparent performance problems by evaluating how many users the system can support based on this 1 second response time guideline. Our results are at best approximate for the purpose, since we do not account for WAN latency when

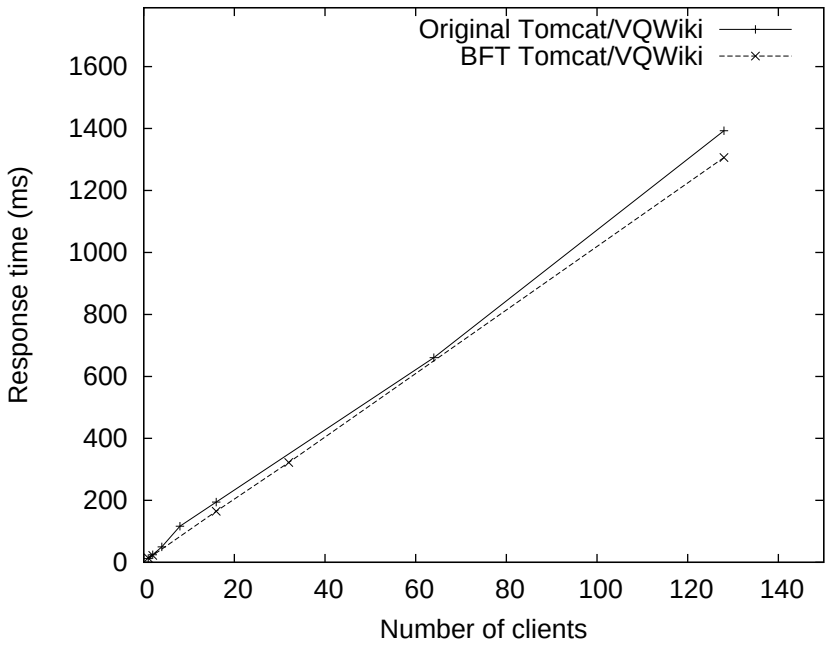

Figure 8.8: Response time vs number of clients for the 90/10 read/write workload - comparison of the original and BFT wiki application

measuring response time. Also, our clients issue requests continuously without any think time between requests. In reality, we expect a larger number of users to be supported with less than 1 second of response time since the think time for human users varies from a few seconds to a few minutes between successive requests. However, our methodology gives us a lower bound on the number of clients we expect the system to support.

We measured the response time of the system while varying the number

of clients making requests to the system. As seen in Figure 8.8, the system is capable of handling over a hundred clients simultaneously while being able to serve requests in under 1 second. While this should be sufficient for a small deployment of the wiki application, it is regrettably insufficient for a large-scale deployment. It remains to be seen whether implementing the improvements outlined in Section 8.2.4, would provide better performance and scale — these optimizations are beyond the scope of this thesis and are avenues for future work.

8.2.6 Summary

In summary, our performance results show comparable performance for write-heavy workloads and degraded performance for read-heavy workloads. This is not surprising, given that we sacrifice parallelism in the server. However we identify ways to improve performance by re-introducing multi-threaded read request processing. While we have not evaluated this approach on the VQWiki application due to its lack of side-effect-free read-only requests, we believe that it would show results that are comparable to the baseline implementation of Tomcat and VQWiki. Another possible improvement for large wiki sizes is the use of file-system snapshots, which is beyond the scope of this thesis, but presents itself as an promising option to explore in future work.

Chapter 9

Future Work

9.1 Multi-tiered services

The UpRight library has been successfully applied to bring BFT to the Tomcat servlet container and the VQWiki web application. While this is by no means a complete argument for the feasibility of BFT web applications, it is definitely a significant step in that direction. The VQWiki web application example used in this thesis utilises the file system to store all its working state and is thus a two-tiered application with the client and the server being the two tiers.

In general, however, web-based systems are implemented as multi-tiered applications [28] with each tier implementing its own fault tolerance scheme for high availability. Take for example a typical multi-tiered web based system consisting of a web application server that handles requests from clients, and a database server that in turn processes requests forwarded to it by the web application server. For simplicity a single database service is considered as a second stage of Execution, but the concepts apply equally to software architectures with more tiers.

Since the working state of the web application is now managed by

multiple tiers (the web application server and the database), applying end-to-end BFT is more complex than doing so for an application in which only one tier stores the application's working state. One solution is to make each tier BFT individually [25]. While this would satisfy the required properties of the system, the cost of applying a BFT protocol at every tier might add a significant performance overhead to the entire system and is also likely to be harder to implement.

An alternate approach is to treat each tier as part of a multi-stage Execution. The first stage of the Execution will serve as a client for the second stage. The protocol between the first and second stage of the Execution will then be the same as the protocol between the Order and Execution stages of the original UpRight library [7] with a couple of modifications. Instead of the responses of the second-stage Execution being sent back to the client, they are instead sent to every first-stage Execution node. The first-stage Execution nodes then accept the response only if the corresponding reply certificate [42] is complete.

This protocol raises an interesting issue for checkpointing — the second stage Execution nodes now also need to agree on their checkpoints and thus need to utilise the Order nodes to do this. However, they do not directly communicate with the Order nodes, but instead do so through the first-stage nodes. For this purpose, the same client voting logic as applied to Execution responses in the original UpRight library [7] can be applied to the checkpoint tokens generated by the second stage when they are received by each of the

first stage Execution replicas. A certificate of these second-stage checkpoint tokens forms part of the checkpoint token returned by the first-stage Execution, thereby ensuring that a stable checkpoint at the first execution stage automatically implies that the corresponding checkpoint at the second Execution stage is already stable.

With the protocol between the two Execution stages in place, it is not difficult to extend this to multiple stages of Execution. Also, the inter-stage Execution protocol can be implemented as an extension of the UpRight library for easy conversion of each of the Execution stages. It is hypothesized that this approach will have a lower performance cost and will be easier to implement than if BFT was added each of the application tiers individually. This, however, remains an open area for future exploration.

9.2 A general framework for BFT Web Applications

We have implemented BFT on the Tomcat servlet container and the VQWiki web application. In doing so, we have created a Byzantine fault tolerant servlet container that in effect does for the web application what the UpRight library does for the servlet container — provide an abstraction of extra functionality necessary to implement BFT. The BFT Tomcat implementation makes it even simpler to implement a BFT web application by implementing incremental delta checkpointing through request logging. Thus, all that is left for the web application to implement is a means of returning a checkpoint of its working state. We hypothesize that it would be easy to develop BFT web

applications by reusing the same BFT implementation of Tomcat. We aim to test this hypothesis by implementing other web applications on top of BFT Tomcat.

Chapter 10

Conclusion

The goal of this thesis is to evaluate the viability of Byzantine fault tolerant (BFT) web applications. It achieves this goal using the UpRight library and modifying the Apache Tomcat servlet container and the VQWiki web application. In doing so, it provides a framework for building more BFT web applications on top of the Apache Tomcat servlet container. While this approach is not expected to be applicable for all web applications, it is expected that it can be applied to most web applications that meet the requirements of the framework, as demonstrated by the simple conversion of the VQWiki web application on Apache Tomcat.

Understandably, there will be a trade-off between the performance penalty of applying such fault tolerance, and the savings in avoiding costly system failures. It is believed that whenever the balance of this trade-off tilts in favour of better fault tolerance, there already exists a low-cost solution, as shown by this thesis.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. 20th SOSP*, Oct. 2005.

[2] T. Abdollah. Lax outage is blamed on 1 computer. Los Angeles Times, 2007.

[3] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5):448–465, 2006.

[4] M. Calore. Ma.gnolia suffers major data loss, site taken offline. `http://www.wired.com/epicenter/2009/01/magnolia-suffer/`, Jan. 2009.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, Feb. 1999.

[6] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26 th annual ACM symposium on Principles of distributed computing (PODC07)*, pages 398–407, New York, NY, USA, 2007. ACM.

[7] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[8] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[9] D. Coward and Y. Yoshida. Java(TM) Servlet API Specification Version 2.4. Specification, Sun Microsystems, Inc, 2003.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. 7th OSDI*, Nov. 2006.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store . In *SOSP '07: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[13] R. Friedman and A. Kama. Transparent fault-tolerant java virtual machine. *Reliable Distributed Systems, IEEE Symposium on*, 0:319, 2003.

[14] Apache Geronimo. http://geronimo.apache.org/.

[15] Glassfish - open source application server. https://glassfish.dev.java.net/.

[16] Hadoop. `http://hadoop.apache.org/core/`.

[17] Apache HttpComponents - the HttpClient library. `http://hc.apache.org/httpcomponents-client/index.html`.

[18] Java web start technology. `http://java.sun.com/javase/technologies/desktop/javawebstart/index.jsp`.

[19] Jetty WebServer. http://jetty.mortbay.com/jetty/.

[20] JavaServer Pages technology. http://java.sun.com/products/jsp/.

[21] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin. Zyzzyva: speculative byzantine fault tolerance. *Communications of the ACM*, Nov. 2008.

[22] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Conference on Dependable Systems and Networks, DSN'04*, June 2004.

[23] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[24] Apache lucene. `http://lucene.apache.org/java/docs/index.html`.

[25] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-fault-tolerant middleware forweb-service applications. *Reliable Distributed Systems, IEEE Symposium on*, 0:131–142, 2005.

[26] R. B. Miller. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, New York, NY, USA, 1968. ACM.

[27] Mozilla Firefox. http://www.mozilla.com/firefox/.

[28] Tiered distribution, version 1.0.1. MSDN Library. [online] `http://msdn.microsoft.com/en-us/library/ms978701.aspx`.

[29] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 03), DCC Symposium*, pages 425–434, San Francisco, CA, June 2003.

[30] J. Postel. Transmission control protocol. RFC 793, Internet Engineering Task Force, September 1981.

[31] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.

[32] Java Servlet technology. http://java.sun.com/products/servlet/.

[33] A. S. Team. Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008.

[34] Apache Tomcat. http://tomcat.apache.org/.

[35] M. van der Kleijn, G. Cronin, M. Teodori, and T. Howland. Vqwiki documentation for release 2.8.1. http://www.vqwiki.org/docs/vqwiki-book.html, February 2006.

[36] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, Oct. 2007.

[37] Vqwiki issue: Update to non-customized lucene version 1.3. http://issues.vqwiki.org/browse/VQW-206.

[38] Vqwiki. http://www.vqwiki.org.

[39] Wikipedia database size. http://stats.wikimedia.org/EN/TablesDatabaseSize.htm.

[40] Winstone servlet container. http://winstone.sourceforge.net/.

[41] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical bft. Technical Report TR24-09, University of Massachusetts, 2008.

[42] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, pages 253–267. ACM Press, Oct. 2003.

[43] Opensolaris ZFS. `http://www.opensolaris.org/os/community/zfs/`.

[44] Zookeeper. `http://hadoop.apache.org/zookeeper/`.

Vita

Rohan Francis Rebello was born in Madras (now Chennai), India to Santosh and Corinne Rebello, joining his big sister Sonia who has never let him forget she is just that. When he got his first computer in 1993, he was fascinated by it but had no idea how it worked.

Curiosity got the better of him, and after graduating from Don Bosco High School in Chennai, he decided to join the College of Engineering, Anna University. Four years later and with a Bachelor's degree in Computer Science and Engineering, he chose to put his knowledge to use at Trilogy Software, Bangalore.

After two years of enjoying the comfortable life of well-paid employment, unable to ignore his restlessness to learn more, he came University of Texas at Austin for a Master's degree in Computer Science. For now, the restlessness has subsided, but it might resurface in the future.

Permanent address: #12, Block 13, SBM Colony
Srirampura IInd Stage
Mysore - 570 023
India

This thesis was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.